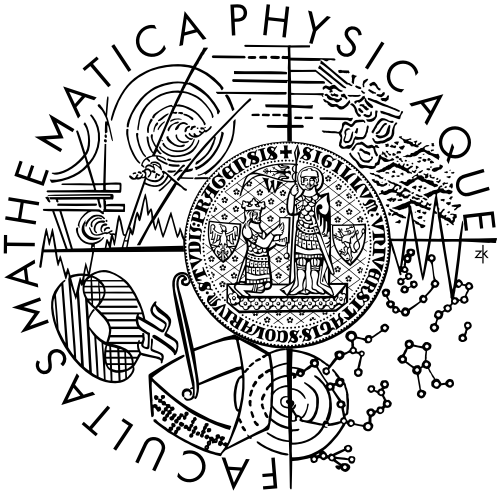
Charles University in Prague
Faculty of Mathematics and Physics

# MASTER THESIS




Tomáš Kohan

# Presenting results of software model checker via debugging interface

Department of Software Engineering

Supervisor of the master thesis: RNDr. Ondřej Šerý, Ph.D.

Study programme: Computer Science
Specialization: Software Systems

Prague 2012

I would like above all to thank my supervisors, RNDr. Ondřej Šerý, Ph.D. and RNDr. Tomáš Poch, Ph.D., for their guidance in shaping my research, their continuous encouragement and major advices.
I am very grateful to all the people who supported my work by their help and advice too.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, August 1, 2012 Tomáš Kohan

Název práce: Prezentace výsledků kontroly softwarového modelu skrz ladící rozhraní
Autor: Tomáš Kohan
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Ondřej Šerý, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Cílem této práce je navrhnout a implementovat nové ladící rozhraní programu Java PathFinder. Vhodným prostředím pro toto rozhraní byl zvolen vývojový nástroj Eclipse. Vytvořené rozhraní graficky vizualizuje výstupy programu JPF a jednotlivé detaily stavu pozastaveného virtuálního stroje (JVM), zvláště pak seznam proměnných a jejich hodnot. Za tímto účelem jsou vytvořeny dva podprojekty, a to debug4jpg a JPFDeb.core. Projekt debug4jpf kontroluje a komunikuje s instancí JPF. JPFDeb.core pak ve formě zásuvného modulu pro Eclipse poskytuje takové uživatelské rozhraní, které je podobné standardnímu rozhraní ladícího programu pro Javu. Oba projekty mezi sebou komunikují přes ad-hoc komunikační protokol, který byl navržen pro tento účel.

Klíčová slova: Java, verifikace, kontrola modelu, JPF, ladící rozhraní

Title: Presenting results of software model checker via debugging interface
Author: Tomáš Kohan
Department: Department of Software Engineering
Supervisor of the master thesis: RNDr. Ondřej Šerý, Ph.D., Department of Distributed and Dependable Systems

Abstract: This thesis is devoted to design and implementation of the new debugging interface to the Java PathFinder application. As a suitable interface container was selected the Eclipse development environment. The created interface visualizes results of JPF and details of paused JVM state, especially a list of variables and their values. Two subprojects were created, i.e. debug4jpf and JPFDeb.core. The first one is responsible for controlling and communication with the JPF instance. The latter one is an Eclipse plugin and provides user interface which is similar to the interface of standard Java debugger. These two components communicate with each other by using the ad-hoc communication protocol created for this purpose.

Keywords: Java, verification, model checker, JPF, debugging interface

# Contents

# Introduction

Nowadays, a variety of applications control important machines and processes in our lives, e.g. surgical robots, spacecrafts, finance, army etc. Any, even small, error can cause incalculable losses. Therefore, testing and, especially, verification and model checking is being important part of the application development process.

Java PathFinder (JPF) is one of the well-known model checking software. It is predominantly used in academic environment, but it is gradually spreading to the commercial sector as well.

This thesis is devoted to design and implementation of a new debugging interface to JPF in order to improve and clarify the output of the test for the user. The current version of JPF provides a textual output which cannot be easily interpreted by the user and also not all details which are available in JPF internal data structure are present.

Our approach is to integrate JPF into widely used graphical development environment (Eclipse) and visualize all runtime items which are available in JPF – threads, stack frames and variables.

The first chapter describes the basic principles of debugging and model checking followed by the analysis of possibility to combine advantages of both approaches – debugging via model checker.

JPF, its internals and usage are described in the second chapter. The most of the chapter is devoted to the JPF architecture and the data model, which are used in the proposed approach.

The third chapter lists the used components of the Eclipse Debug framework. The graphical part of our approach is making use of this framework to integrate into Eclipse development environment. The launch framework is used for starting and initialization of the implementation, while debug model and variables components visualize all threads, stack frames and variables.

The actual application design is described in the fourth chapter. The application is divided into two main components, JPFDeb.core and debug4jpf. The JPFDeb.core component is an Eclipse plugin which is responsible for communication with the Eclipse instance. In addition to this, the debug4jpf component communicates with and controls the JPF instance. There is an ad-hoc communication protocol designed and it is used for the communication between JPFDeb.core and debug4jpf.

The fifth chapter lists some of the implementation details and the user guide. There are several sample projects included on the attached CD. The user guide is a step-by-step instructions list of how to run these sample projects, or

other Java projects whose model should be checked by JPF.

To summarize, the main goals of this thesis are:

- Design and implement new debugging interface of JPF as a part of an integrated development environment.
- Visualize paused JVM state and JPF results in order to help the user to inspect the error state.
- Add variables and their values as a part of this visualization.

# Chapter 1

# Model Checker and Debugging

Debugging is the commonly used technique to test and fix applications. Besides many other techniques, there is one, named model checking, whose results give a strong confidence that the application is bug free (considering only those bug types which the model checker has been configured to test). The main goal of this thesis is to combine the well-known debugging approach with the strong results of model checking.

This chapter is focused on three topics. The first one is debugger and debugging generally. The next item is model checker itself and its basic generic principles of its operation. The last part of this chapter addresses the combination of model checker and debugging principles.

## 1.1 Debugging

Debugging a program is one of the first activities that a developer needs to learn. It is a process of executing the program in a special mode, which ensures that the Java Virtual Machine (JVM) communicates with the debugger. Debugger provides the user (developer) with following debugging features:

**Breakpoints** – they are used to suspend the JVM processing at certain point of the execution. It is possible to mark the line in source code as a breakpoint and as soon as the JVM hits this line during the execution, it gets suspended and the user is notified about this situation. The current state of the JVM is displayed in the debugger window (in case of debugger with GUI) or is prepared to be queried (in case of command line debugger). User can investigate the state of the JVM and, if needed, it is possible to step further in the execution.

**View state** – The state of the JVM includes threads, stack frames, variables and variable values. If the user selects a stack frame of a thread, then the cursor, in the source code editor, is placed on the line which is currently being executed in the JVM and all local variables and their values are loaded into GUI. Some debuggers allow the user to modify variable values in the middle of program execution and the following processing is done with the new value of the variable.

**Stepping** – If the thread is suspended, the user can step through the following processing. There are multiple step variants. *Step over* means to step to the next line of the source code in the current class and method or to step to the caller of the current method if there is no following line available in the current method. *Step into* means to step inside the method that is going to be called on the current line of source code. The processing is suspended at the beginning of the method that is being called. *Step return* means to step to the caller of the current method. The current method is executed to the end and as soon as the processing comes back to the caller, it is suspended. The last option is to let the processing continue in its execution till the end or next breakpoint (if available).

These features let the user investigate the program behaviour during the execution and help to understand the reasons why it behaves that way.

An example of a situation, when it is suitable to use the debugger, is the program that crashes with an NullPointerException exception. From the stack trace, that is printed to the error output stream, it is obvious in which class and on which line this exception occurred. But it does not have to be obvious which variable value is null (e.g. stringBuffer.append(myClass.toString()) – stringBuffer and also myClass can be null) and, more important, it is not clear how we got to the error. If the user marks this line with a breakpoint and runs the program again, the execution stops on this line before the exception is thrown and the user can investigate in variables view, which of these two variables (stringBuffer or myClass) is null and in the list of stack frames how the program got to the error state.

## 1.2 Model Checker

Generally, model checking is a process of program correctness verification. At the beginning of this verification process, a tester has to define assertions and validation properties that should be validated in the program model, e.g. not deadlocked property, no uncaught exception etc. The model checker than validates that the assertions and validation properties are valid in whole program model regardless of execution trace. Different execution traces differ in thread interleaving (thread schedule), variable values, external systems interactions etc.

The goal of a model checker is to traverse the complete execution space of the program model and verify that none of the verification properties is broken on any execution trace. This is the main difference between model checker and standard debugging. If the program is debugged, it confirms that there is no error on the execution trace that was debugged. In case of model checker, after the verification is done, it confirms that there does not exist any error of defined type in the program at all.

Model checking can be executed on different levels of program model abstraction. In some approaches, a simplified program model is generated from the tested program first. Final verification is then not executed on the tested

program itself, but on the simplified model. On the other hand, some of the approaches traverse the program model by executing/simulating the program itself. One of such approaches is Java PathFinder. It simulates Java virtual machine in order to execute the tested program.

Naturally, in order to avoid traversing the program model in cycles, the model checker needs to store information about what has been already executed. This information is stored as a list of already visited states. In case the model checker reaches a state that has been already visited, it backtracks to the latest state on the execution trace that has at least one not explored execution trace option. These options are created in case there is a possibility to schedule threads differently, change variable value, or anything else that would have an impact on execution trace. It is obvious that the number of states can be huge and it is necessary to search for particular state quickly. Several techniques, how to store states efficiently, have been introduced – symmetry reduction [1, 2, 3, 4], abstraction [5, 6, 7, 8, 9], static and runtime analysis [10].

The result of a model checker is usually a execution trace that leads to a bug that has been identified. The form of the result can be different – Java Virtual Machine snapshot, a generated program that simulates the tested program execution on the trace that leads to the error state, etc.

Usually, there is no user-friendly user interface provided by the tools or if it is provided then it is tool-specific user interface which the user do not know and need to learn how to use it. In comparison, Java debugger's user interface is well-known by all Java developers and testers. It would be beneficial for the user if the testing power of model checker could be combined with the user-friendliness of standard Java debugger's user interface.

## 1.3 Debugging via Model Checking

The two previous sections deal with model checking and debugging. In order to create a debugging interface for a model checker it is needed to understand what "debugging via model checking" means, at least in context of this thesis. As stated above, usually a model checker returns a description of program state when an error occurred. At this point, the model checker knows exact trace from the beginning directly to the error. It would be really helpful for the user to let the model checker find the trace to the error and then use standard well-known debugging features to investigate this trace, find the root cause and fix the problem. This is short, high level description of the benefit that would be introduced by possibility to debug via the model checker.

Figure 1.1 shows the sample situation when the debugging capability would help the user to understand the root cause of the error, which was found by the model checker. As the first step, the model checker traverses through whole program model and identifies the trace that leads to an error (marked as 1-4 in the picture). As soon as there is the trace available, the debugger would be used to explore the error trace, e.g. put a breakpoint to point 2 and check variable values in case that the processing hits this breakpoint.

Current versions of usually known model checkers either do not support

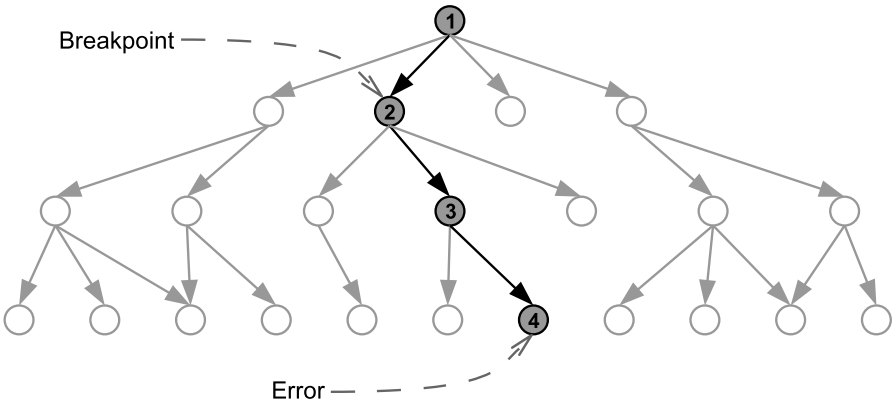



Figure 1.1: Benefits of debugging via model checking

such debugging capability or their focus is different than the model checkers which are in interest of this thesis (e.g non-Java model checker – [11], custom made graphical user interface – [12]). The solution for this is to create a debugging interface for one of the model checkers and make use of this interface in one of the well-known integrated development environment for Java. The generic diagram with interactions between components is depicted in Figure 1.2.

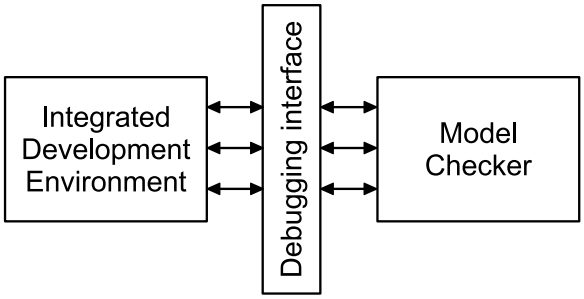



Figure 1.2: Generic component diagram with interactions

In this approach, the model checker is not a standard Java Virtual Machine (JVM), however it acts like a JVM. Therefore, not all debugger features may be available. It can be (1) not possible to use such feature in this context, (2) it would have no positive impact on the execution or (3) it would be so unpredictable and non-intuitive for the user, that it is better not to support such feature.

1. The features that need to be supported:

   **List of threads** – show the list of current threads and their status in order to let the user know what is being currently executed in model checker.

**List of stack frames** – show list of stack frames for each thread in order to let the user know what is the sequence of method calls which led to the current state.

**Variable names and values** – show list of variables and their values for each stack frame, that may have significant impact on the program functional behaviour.

**Source code lookup** – link each stack frame with the source code it refers to, that makes it easy for the user to get to the source code and check the implementation without the necessity to search for the particular file himself/herself.

**Breakpoint** – suspend the execution if the breakpoint is hit, which is helpful in case the user needs to check the system state even before the error occurred.

2. The features that could be supported, but it would have no positive impact on the execution:

   **Variable value change** – changing the variable value in runtime could cause the trace not to finish in the error state

3. The features that could be supported, but it might be too confusing for the user what it means in the context of the model checker:

   **Stepping** – no possibility to step one thread (as usual in Java), thread schedule is predefined for the trace and therefore all threads would need to step

The more detailed arguments for dividing the debugging features into these categories are listed further in the text.

# Chapter 2

# Java PathFinder

Java PathFinder, a well-known model checker, is used as the target model checker for this thesis. The reason for this is its widespread use in academia as well as industry. JPF is unique compared to other model checkers because of its configurability and extensivity. Many projects have been developed as JPF extension projects with many different purposes, e.g. jpf-aprop [13] (Java annotation based properties and related checkers) and jpf-trace-server [14] (provides storing, querying and analysis of the execution trace). All of them make use of JPF's configurability and extensivity.

Java PathFinder is a piece of software written in pure Java that behaves like a Java Virtual Machine with the difference, that it is not executing only one execution trace, but ideally all possible execution traces. Word "ideally" is used because the number of such traces could be enormous and therefore JPF must implement techniques like flexible heuristics and state abstractions. During the tracing, JPF is searching for various types of defects. Most common ones are deadlocks and unhandled exceptions. Deadlocks are especially hard-to-find type of concurrency defects in highly multi-threaded applications.

Java PathFinder can be run either as standalone command line tool, or as embedded into applications like development environments.

NASA Ames Research Center started with the research in the area of model checking in 1999. The result of this effort was Java PathFinder. JPF is described in [15] as follows:

> *"The answer used to be simple: JPF is an explicit state software model checker for Java bytecode. Today, JPF is a swiss army knife for all sort of runtime based verification purposes."*

The result of JPF is one or more (depends on the configuration) execution traces which lead to defects. The advantage over the standard Java debugger is that JPF returns whole execution trace – from the beginning of the application directly to the defect. For example, if a deadlock is found in a multi-threaded application, the trace contains the exact thread schedule which leads to the defect.

## 2.1 JPF Capabilities

Searching for deadlocks and unhandled exceptions (e.g. `ArithmeticException` and `AssertionError`) is standard feature provided by JPF. This capability can be further extended by custom property classes and listener-based extensions, which implement other property checks (e.g. race conditions).

In general, JPF can be used to model check any Java program with two limitations – native methods and size.

Currently, there is only a very limited support of platform specific, native method calls. JPF mechanism, called Model Java Interface (MJI), enables creation of special library versions which contain native classes capable of execution in JPF environment. Anyway, there is no support in JPF for `java.awt` and `java.net` packages. The `java.io` package is supported with limitations.

The existence of tested program size limitation is expected (with regards of the model checker internal algorithms), however it can be hardly quantified. Some sources, [10, 15], indicate that the program size limit is somewhere at 10,000 lines of code. It is quite strong statement, as it is possible to create program with 200 lines of code which cannot be tested by a model checker and vice versa. The size limitation is strongly dependant on the program internal logic, especially, on the number of states created during the model checking.

Due to the limitations above and JPF's capability to explore complete thread scheduling space, JPF is mainly used to verify multi-threaded Java programs.

## 2.2 JPF Architecture

The JPF architecture is tightly coupled with two major abstractions – JVM and Search (Figure 2.1).

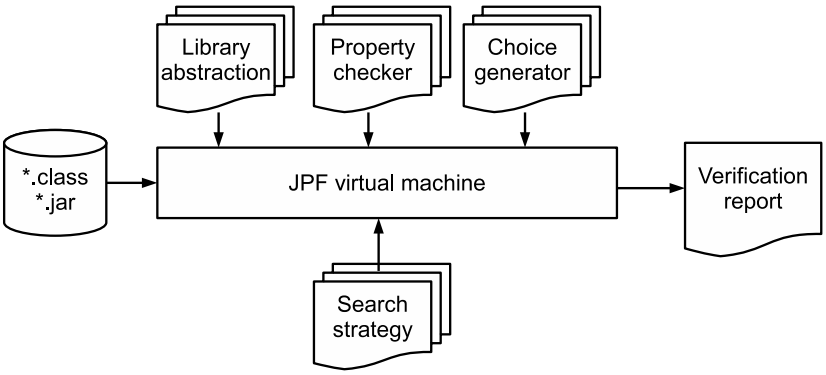



Figure 2.1: JPF components

### 2.2.1 JVM Object

In JPF, the JVM object is a state generator which is responsible for state management. It generates states during the execution. A state is a set of meta information which fully describes the current JVM snapshot. These states need to support several operations which have to be executed quickly. The performance requirement arose from the fact that the number of states can increase to millions. The check for equality is important for identifying already visited states. In such case, it make no sense to continue with this execution trace and the execution can be backtracked to previous state that has unexplored execution trace. In order to support this procedure, states have to support following operations:

- check for equality
- queried
- stored
- restored

The JVM and Search objects collaboration is handled by three JVM functionalities:

- *Forward* – proceed to next state on execution trace
- *Backtrack* – restore previous state on the execution path
- *RestoreState* – restore previously stored state

### 2.2.2 Search Object

The Search object is responsible for deciding from which state should the execution proceed – ask JVM to either generate a next step (forward) or return back to the previously generated one (backtrack or restoreState). Although the JVM object is the state manager, it only complies with the requests from the Search object.

The next very important feature of the Search object is configuration and evaluation of property objects. The standard JPF release contains a simple depth-first search (`DFSearch`) and a priority-queue based search (`HeuristicSearch`) which can be configured to use miscellaneous techniques for choosing the most suitable state in case there are multiple successors of the current state. The Search object iterates through the state space until all states are explored (all relevant traces have been explored), or it finds a property violation.

## 2.3 Listeners

There are many potential situations when the standard Search-JVM collaboration is not enough and the user would need more, e.g. gather statistics, influence the state exploration sequence or fetch variable values. These are the typical tasks that are implemented as JPF extension projects. In context of this thesis, it is a debugging interface and graphical user environment. One of the main features of JPF is supporting such extensibility without the necessity of modification of neither Search nor JVM object implementation.

Listener mechanism in JPF implements the Listener pattern [16]. The user has to register his/her listener instances with the JVM and the Search object (Subject). As soon as the instances are registered, they are notified every time the Subject performs a certain operation. The listener has the opportunity to get all necessary details that are available through the Subject object. The Subject object is always passed as a parameter to the notification handler.

Figure 2.2 lists Search object's basic notifications, their relationships and transitions. In addition, there are several JVM object's notifications which are not relevant in the context of this thesis.

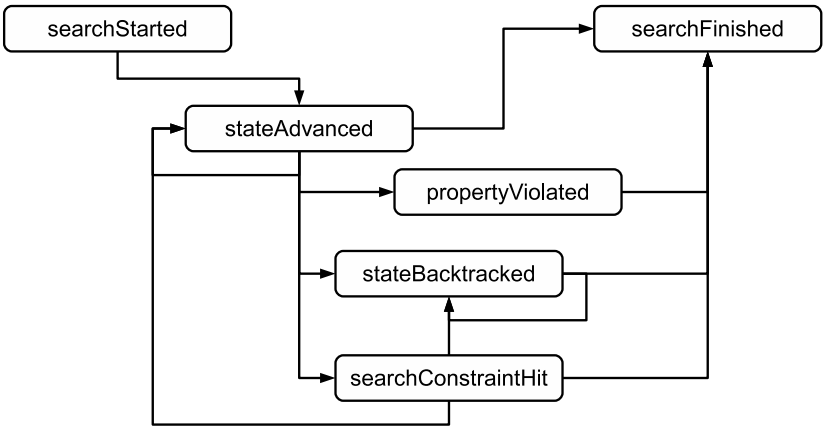



Figure 2.2: JPF notifications

## 2.4 Configuration and Inputs

JPF is an open system and authors want to encourage users to create additional JPF enhancements and extensions. These projects are called JPF components and require good support for parametrization and extensibility. In order to fulfill this requirement, well-defined general configuration mechanism has been designed and implemented in JPF. As many components require configuration classes to be loaded by providing a classname as a parameter, the approach in JPF cannot use a predefined configuration class with predefined concrete fields. This would be too limiting for the components like Search, Heuristic,

Scheduler, and UI implementations. The configuration object which meets the requirements is the one that is:

- based on symbolic values
- extendable at will
- passed down in a hierarchical initialization process

JPF covers this by designing a central directory object which initializes all configurations by loading Java property files in hierarchical structure with five different initialization layers:

1. *Site* (site.properties) - optionally installed JPF components
2. *Project* (jpf.properties) - settings for each installed JPF component
3. *Application* (*.jpf) - the class and program properties that should be checked by JPF
4. *Command line* - overrides any property listed above
5. *Verify API* - programmatic configuration at runtime

The configuration files are loaded in prioritized order, which ensures that the properties loaded in deeper level overrides already existing property with the same name loaded in higher levels.

Besides properties configurations, the most important and mandatory input is program (bytecode) whose model should be checked by JPF. All resources (e.g. class directories, jar files, etc.) needed by the program must be available and accessible on the classpath. Moreover, the main class of the program with main method has to be specified as an input parameter. This class is considered to be the root point from which all execution traces begin.

## 2.5 Output

JPF provides three standard output formats and, thanks to the extensibility options of JPF, it could be extended to support any custom made output format. Each of these formats has its own purpose:

**Application output** – it is mainly connected to `System.out.println(...)` calls in application code. This application logging behaves differently in JPF context as there can be one step (representing a set of lines in source code) executed several times. JPF has the ability of backtracking steps and rerun them in case of different input variable values, etc. This means that one log message can occur several times in the log. It could be confusing for the user and therefore JPF supports two configuration parameters – `vm.tree_output` and `vm.path_output`.

**JPF logging** – it informs the user about what JPF does internally. It supports several levels of detail and is based on standard `java.util.logging` infrastructure. JPF incarnates classes from Java logging framework in order to support logging configuration via JPF configuration and not via system properties. There are two main configuration items – controlling log output destination and setting log levels.

**JPF reporting system** – it provides details of JPF run, e.g. property violation, print traces, show statistics, etc. This is standard out-of-the-box user interface for JPF test results, which supports different output formats (text, XML, API calls) and targets (console, IDE). Hence different applications and projects require different output formats (items displayed and their order), JPF reporting system is able to be configured (by general configuration mechanism) and extended to support these new applications and projects needs.

### 2.5.1 JPF Reporting System

There are output phases predefined during the JPF execution. Each phase has configured and ordered list of supported topics. These phases and topics are part of the JPF Reporting System. Phases currently supported by the system are:

- *Start* – system and application configuration summary
- *Transition* – list of all transitions between states (switched off by default)
- *Property violation* – description of the state where the property violation occurred
- *Finished* – result description and statistics information

The standard *property violation* topics include:

- *Error* – shows the type and the details of the error
- *Trace* – shows execution trace which leads to the error
- *Snapshot* – lists threads details (status and call stack)
- *Output* – shows the program output for the trace
- *Statistics* – shows statistics information

The *finished* list of topics that summarizes the JPF run:

- *Result* – lists overall verification results (errors)
- *Statistics* – shows overall statistics information of the whole verification process

Following sections describe a default output of the JPF which checks the model of one of the sample programs used later in this thesis – TestDeadlock. Particular fragments of the output are depicted with description and proposal of possible improvements.

### Start

Figure 2.3 shows messages which are written to the log right after a JPF run starts. The JPF version and the main class name of the program that is being checked is printed.

```
JavaPathfinder v6.0 (rev 617+) - (C) RIACS/NASA Ames Research Center


========================================== system under test
application: TestDeadlock.java
```

Figure 2.3: The JPF output during start phase of execution

### Error

Figure 2.4 depicts the JPF output in case of an error hit. In this particular case it is a deadlock and thus JPF lists all threads and their states. This part of output informs the user that a deadlock occurred, the user can see the threads which are blocked, however, this information is not detailed enough to let the user know where and why the deadlock occurred. More details can be found in the next output fragment.

```
========================================== error #1
gov.nasa.jpf.jvm.NotDeadlockedProperty
deadlock encountered:
  thread id=1,name=Thread-1,status=BLOCKED,priority=5,
    lockCount=0,suspendCount=0
  thread id=2,name=Thread-2,status=BLOCKED,priority=5,
```

Figure 2.4: The JPF output during error phase of execution

### Snapshot

Figure 2.5 shows the snapshot of the virtual machine state at the time of error hit. Again, there are threads listed, but this time there are more details added like owned locks, lock on which the thread was blocked and current call stack.

The important thing that would help the user to easily understand what happened and how the program got to the error is missing – variables and their values. The execution path of most programs is highly dependant on the

values of variables (e.g. branches which are done based on a variable value, loop iteration index, etc.).

```
====================================== snapshot #1
thread id=1,name=Thread-1,status=BLOCKED,priority=5,
    lockCount=0,suspendCount=0
  owned locks:TestDeadlock$SyncRunnable@141
  blocked on: TestDeadlock$SyncRunnable@142
  call stack:
        at TestDeadlock$SyncRunnable.doSomething(TestDeadlock.java:12)
        at TestDeadlock$SyncRunnable.run(TestDeadlock.java:25)

thread id=2,name=Thread-2,status=BLOCKED,priority=5,
    lockCount=0,suspendCount=0
  owned locks:TestDeadlock$SyncRunnable@142
  blocked on: TestDeadlock$SyncRunnable@141
  call stack:
        at TestDeadlock$SyncRunnable.doSomething(TestDeadlock.java:12)
```

Figure 2.5: The JPF output during snapshot phase of execution

### Result

Figure 2.6 depicts the output fragment with results overview. It is a list of errors that have been found. With the default configuration, JPF finishes after first error is found. If the user changes the configuration to find all errors, JPF will list all errors in error, snapshot and result fragments of the output.

```
====================================== results
error #1: gov.nasa.jpf.jvm.NotDeadlockedProperty "deadlock encountered:
```

Figure 2.6: The JPF output during result phase of execution

### Statistics

Statistics fragment of the output (Figure 2.7) lists statistical details of the execution. The important items in the list are states, search, and max memory. The states and search values give the user an overall picture how complex the state exploration was. Higher numbers mean more complex state space. As mentioned in previous chapter, the memory allocation is important for storing states. The max memory allocation is important from the point how much more complex can the program model be, to be able to check the model by JPF on the current hardware configuration.

### Finished

Finished fragment (Figure 2.8) just informs the user when the actual JPF run finished.

```
====================================== statistics
elapsed time:       00:00:00
states:             new=8, visited=2, backtracked=2, end=1
search:             maxDepth=8, constraints hit=0
choice generators:  thread=8 (signal=0, lock=4, shared ref=0), data=0
heap:               new=347, released=11, max live=347, gc-cycles=9
instructions:       13412
max memory:         15MB
```

Figure 2.7: The JPF output during statistics phase of execution

```
====================================== search finished: 3/31/12 3:05 PM
```

Figure 2.8: The JPF output during finish phase of execution

The key points that should be improved by our approach are user-friendliness and availability of the list of variables for each thread and their stack frames.

The user-friendliness means to provide the user with a graphical user-interface (GUI) which will be integrated into the widely spread IDE. It is always beneficial to visualize information of such level of detail, so the user does not need to study the text information and realize all the necessary relations and dependencies himself/herself.

As mentioned before, variable values are important for proper identification how and why the program executed along the trace which ended up in an error state. The proposed user interface will provide all variables which are available for each thread and its stack frame. This lets the user easily identify the error cause.

# Chapter 3

# Eclipse Debug Framework

The Eclipse Debug framework is part of the Eclipse development environment by default. It is a powerful API that helps developers to integrate new debuggers into Eclipse. The debugger can use any programming language and the developer various options how to modify the debugger behavior to fulfill the requirements. This chapter presents an overview of the Debug framework as was introduced in [17] and is more focused on the topics that are required and used within our approach.

## 3.1 The Launch Framework

The Debug Framework provides the environment for running or debugging a program. The process of configuration, initialization and execution of the program is called the *launch*. The *launch* in Eclipse has the power to run or debug the program within Eclipse. This activity is performed by *Launcher*. It is a set of classes loaded into Eclipse as a plugin, which is able to execute a program, e.g. Java program, JUnit test, web application, etc. In context of this thesis, it will be responsible for running a model checker (JPF) to check a Java program.

In Eclipse 2.0, a rich API for building plugins with launch behaviour is provided to plugin developers. Launching is centered around two main entities

- launch configuration type
- launch configuration

A *launch configuration type* is registered in *launch manager* which is a central point for all launches. It has its own specific attributes which are stored in launch configurations.

Each *launch configuration* is of a certain type, e.g. JUnit test, web application. A launch configuration is a persistent map of keys and values. Launch configuration attributes, which are necessary to launch a program, are stored in this map. Two types of launch configuration exists – *launch configuration* and *launch configuration working copy*. Configuration items are persisted with-ing launchConfigurationWorkingCopy, while launch configuration is a kind of

configuration template. In case a new launch configuration is needed, a launch configuration (template) is cloned into launchConfigurationWorkingCopy and the user modifies and store this working copy. Launch configurations are meant to be shared across different launch modes – run, debug, profile (launch modes currently supported by the Debug platform, however, the list can be extended).

The lifecycle of the launch configuration is handled by launching infrastructure. As soon as the plugin developer provides launch configuration entity implementation, the Debug framework core plugin manages creation and persistence of the configuration. The user interaction with launch configurations is supported by the Debug framework UI plugin's *launch configuration dialog* and other launching-related UI elements. The content of this dialog is defined by a tab group. A *launch configuration tab group* is a set of tabs, which are used to display and edit a single launch configuration.

*Launch delegate* is bound to an existing launch configuration type for a specific launch mode. The developer is supposed to sub-class the abstract launch delegate provided by Debug framework which is responsible for:

- building the set of related projects before launching
- searching for errors in all related projects and aborting the launch in case of any errors
- searching for unsaved editors before launching and allowing the user to save them

The user application, which is being debugged, is executed via `IProcess` implementation, which is based on `java.lang.Process`. Each process, which is added to the launch, may provide the standard I/O streams which are redirected to consoles in the UI.

## 3.2 Debug Model

The debugger works with a representation of the system and not with bits and bytes. This representation is called Eclipse Debug model in Eclipse Debug framework. It consists of *debug model elements* (Figure 3.1) such as debug target (`IDebugTarget`), thread (`IThread`), stack frame (`IStackFrame`), variable (`IVariable`), variable value (`IValue`), etc. The Eclipse Debug UI interacts with these objects and displays the graphical representation of them in Eclipse environment. The plugin developers provide the implementation of these interfaces.

Besides the debug model elements, the debug model contains several objects which do not represent the model element itself, but rather its *capabilities*, *events*, or *actions* which can be performed with the element.

The standard capabilities provided by Eclipse Debug Model elements are Terminate (`ITerminate`), Suspend and Resume (`ISuspendResume`), Step (`IStep`), Drop to Frame (`IDropToFrame`), and Disconnect (`IDisconnect`). The standard debug elements implement standard capabilities –

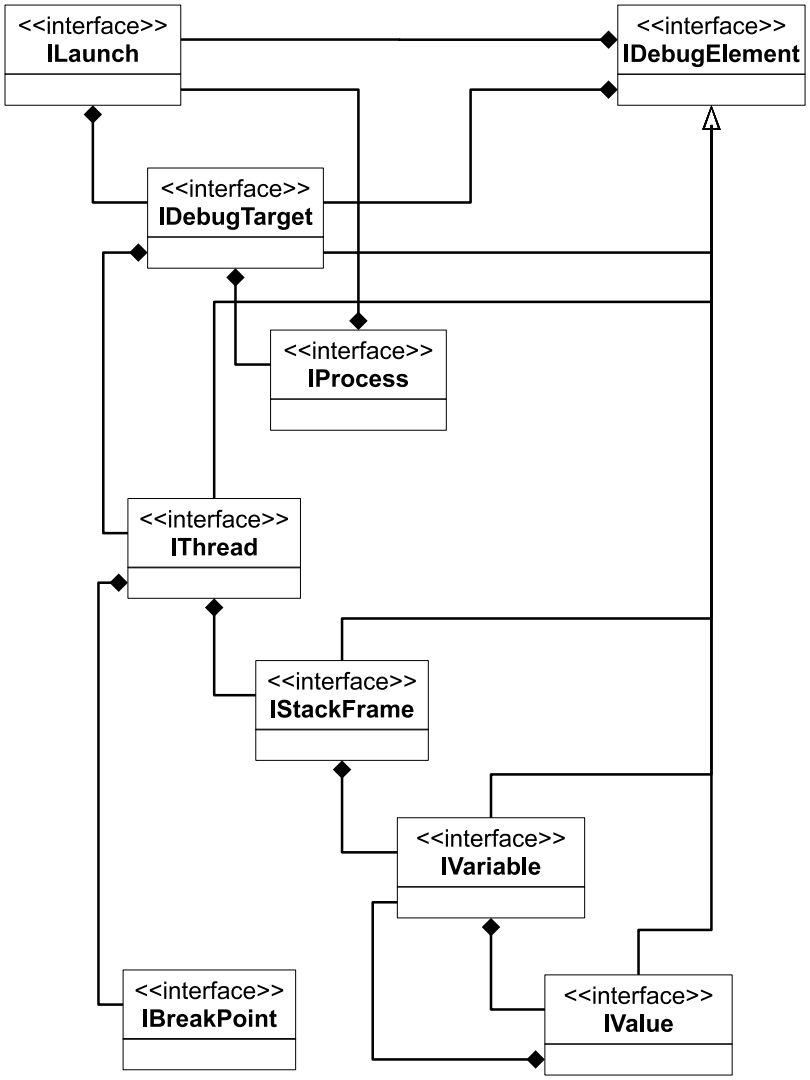



Figure 3.1: Eclipse Debug framework data model elements interfaces

```
IDebugTarget extends ITerminate, ISuspendResume, IDisconnect
IThread extends ITerminate, ISuspendResume, IStep
IStackFrame extends ITerminate, ISuspendResume, IStep
```

The user interacts with these interfaces by using the debug toolbar buttons and/or actions.

The debug model elements and process implementation is supposed to generate debug elements which are required by the user interface (their list is specified in the `DebugEvent` class). A debug event represents something that has happened in a program that is being debugged or run. Each event contains a type identification and detail code, which specifies a reason why this event occurred, e.g. suspend could be caused by step end, breakpoint, client request, expression evaluation (with side effect in the debug target), or expression evaluation implicitly (no side effect in the debug target).

The user interface reacts to debug events by updating the *debug views* which display debug elements. The list of standard debug views contains views such as Debug, Variables, Registers, Expressions, Breakpoints, and Console. One of the behavioral characteristics that supports user-friendliness is that debug views try to keep the selection and expansion state of tree views based on debug element equality in case the view is refreshed. If a debug element remains stable, the view will not change too. Text and images (icons) are used by debug views to visualize debug model elements. The standard images, that are provided by the platform, can be overridden by custom ones by using custom *debug model presentation*.

The debug actions are a little like counterpart to the debug events. If the user changes the selection or clicks one of the action buttons in the user interface, it raises the debug action. The standard debug actions are step, suspend, resume, terminate, disconnect, and drop to frame. Some of these actions can be raised also as a reaction to the debug event.

## 3.3 Breakpoints

The user can suspend the execution of the program by adding breakpoints at selected locations or upon a defined condition. There are several types of breakpoints – line breakpoint, watchpoint, run-to-line, and exception trap. The Debug framework provides operations for working with breakpoints, i.e. notification if breakpoint is added, removed, or changed; breakpoints persistence; and temporary disabled breakpoints. The list of supported types of breakpoints depends on the capabilities provided by the underlying architecture and implementation.

The *breakpoint* is a debug model element which implements `IBreakpoint` interface. In addition, the Debug platform provides `ILineBreakpoint` and `IWatchpoint` interfaces too. The breakpoint implementation is self-sufficient, which means that it has all needed details for registering itself into the debug target and it also provides a default constructor which is used during environment startup for creation of stored breakpoints. Thanks to the interface, the platform is able to provide management of breakpoint state and breakpoint attributes persistence by default. On the other hand, the behaviour of the breakpoint is fully dependant on the implementation, e.g. custom properties, installation, etc.

All breakpoint attributes (e.g. hit count, suspend policy, enabled, condition) are stored in *markers*. The Debug platform provides `IMarker`, general marker for breakpoints (and other elements such as bookmarks, compilation errors, etc.) in files. It is capable to display these elements in editor's vertical ruler. Basically, the marker is a list of key/value pairs of primitive data types which holds all breakpoint attributes. Each breakpoint is connected with the corresponding marker which is responsible for storing its attributes and display itself in the editor. Breakpoints locations are shown in the *editor*. The Debug framework provides `AbstractDecoratedTextEditor` which displays a ruler for visualizing markers associated with the file that is loaded into the

editor. User defined editors can subclass this abstract one in order to support the ruler feature. The ruler can be double-clicked and it can fire more than one action depending on the context. It can set a line breakpoint (line or method entry) or set a watchpoint.

All breakpoints are managed by *breakpoint manager*. It is a repository of all breakpoints in current workspace, which is responsible for registration/removal of breakpoints as they are created/deleted. In addition, it generates change notifications for all changes that were done on breakpoints (add, remove, attribute change). If class implements `IBreakpointsListener` interface and is registered in breakpoint manager, it gets notified of all change notifications that were generated.

In context of breakpoints, the *debug target* installs breakpoints. As soon as the debug target is instantiated, it queries the breakpoints manager for relevant breakpoints. These breakpoints (called deferred breakpoints) are then created and associated markers are loaded into editors and underlying environment. The debug target is implementing `IBreakpointListener` and therefore it gets notified if some breakpoint is changed. In such case it updates the breakpoint in the underlying runtime.

The common set of breakpoint types, i.e. line breakpoints, method breakpoints and watchpoints, are supported by most debuggers. These breakpoints are created by *global retargettable actions* which are supposed to prevent from adding new actions into menu with very similar name and/or function and to preserve common look and feel.

## 3.4 Source Lookup Framework

Current debuggers provide many advanced features that help the user to use the tool. Highlighting the currently executed line or statement in source code is one of them. It became a "must have" feature for current debuggers. The debugger has to identify and find sources for binaries that are being executed. This means looking for the file along a path of directories, zip archives, Java archives, etc.

The Debug framework provides an implementation of a standard "search along a path" type of *source locator*, which consists of director, participant and container.

A *source director* keeps the ordered list – the "path". The Debug framework provides the default implementation of source director, which considers the "path" as consistently ordered sequence of source containers. In addition, the source director is able to compute a default source path based on launch configuration type, unless a source path is explicitly defined by the user.

A *source participant* connects stack frames with file names and a container searches for files by filename.

The platform provides a set of default *source containers*, i.e. workspace folders, projects, archives and local file system directories and archives.

## 3.5 Variables

Variables and their values are listed in the *Variables view*. This view capabilities include listing variables and their values, highlighting variables that have changed their values, and following the "logical structure". The logical structure is usually more appropriate to display, if it is compared to raw structure. For example, if there is a variable of type list, user is usually not interested in technical implementation details (linked list, array, etc.), but rather wants to see elements of the list in the form of an ordered list.

# Chapter 4

# Application Design

This chapter describes the approach which improves the user interface (UI) of JPF. As described in the previous chapter, UI of JPF is far from user-friendly and easy to use. It requires a deep knowledge of Java threads, Java scheduling and assertions, but it still takes a while to realize where the problem is. This thesis proposes new debugging interface to JPF. By utilizing Eclipse Debug framework in Eclipse Integrated Development Environment (IDE), JPF can be used to identify critical faults of the tested software and it can be easily debugged via standard debugging interface.

The main idea is to detect potential problems with JPF and afterwards stop the application execution at the point which JPF identified as problematic. This gives the user clear view where the problem is and what was the current state of the application when the problem occurred. All this information is clearly displayed in Eclipse via standard Java application debugger interface.

The original proposal to use JPDA as a debugging framework was changed to Eclipse Debug Framework. The main reason for this decision was the fact that JPDA is fully-featured "heavy" debugging framework for Java environment and the complexity of implementing all of its interfaces would add features that are not goals of this thesis.

Such advantages would be possibility to use this solution with other JPDA compliant debuggers (however the compatibility is not guaranteed as there might be some implementation differences and JPF is not JVM for which JPDA was designed). Another advantage would be remote debugging which is currently not available in our approach. However, the design of the solution and the ad-hoc communication protocol are open for any type of extensions to support remote debugging.

Disadvantages of using JPDA would be complexity and cumbersomeness of design and implementation. JPDA uses low level byte stream communication. It supports many features that are not applicable and/or not available in JPF environment and the implementation would be inadequately difficult or even not possible (e.g. variable value change in runtime, watches).

On the other hand, Eclipse IDE is widespread across Java developers and the solution is simple, readable and fulfilling all goals of this thesis.

Advantages of using Eclipse Debug Framework are:

- Simplicity and readability – Implementing the debug model and calling well-defined Java API make the solution easy to implement, read and understand without any workarounds.
- Possibility to propose communication protocol that fits best for JPF and Eclipse integration.
- Full compatibility with Eclipse – the most common Java development environment (based on research report [18] - Eclipse has 65% of market share).

Drawbacks of using Eclipse Debug Framework are:

- Usage of ad-hoc communication protocol
- Tight-coupling with Eclipse development environment

## 4.1 Application Components

The overall solution components and their interactions are depicted in Figure 4.1. It consists of six main components (implementations of three of them, gray ones, are parts of this thesis):

- Eclipse (Eclipse Debug framework)
- JPFDeb.core
- debug4jpf
- Listener
- JPF
- JPF JVM

Eclipse, JPF and its JVM components are described in previous chapters as they are already existing parts of the solution. Following sections explain the role of the remaining three components and their mutual communication with each other as well as communication with the former three.

## 4.2 The debug4jpf Component

JPF and debug4jpf are in a role of debuggee. The debug4jpf component is the front-man of JPF. It initiates JPF, controls its execution and provides requested details from JPF to the debugger.

The debug4jpf component uses its own JPF listener to control the execution of JPF. Every event generated by JPF is forwarded to the listener. These events let debug4jpf follow the progress of the JPF execution.

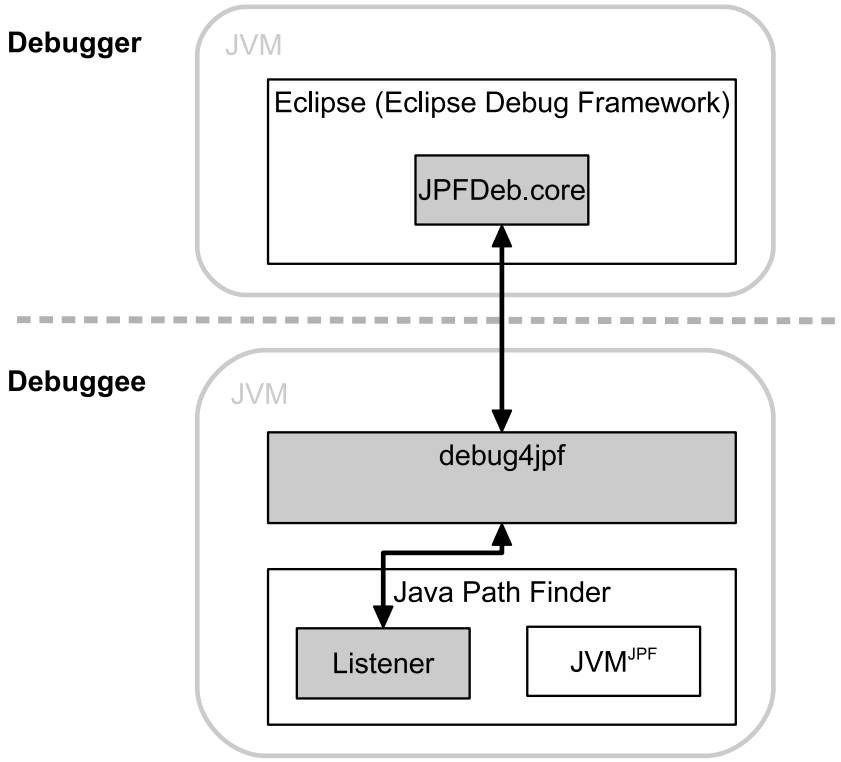



Figure 4.1: Solution architecture

VM listener's events always receive the `JPF` object instance as a parameter. Therefore, debug4jpf has complete information about the actual execution progress, memory content, etc. At the same time, the search listener's events receive `Search` object as a parameter. This object provides complete information about the actual step of the state space exploration.

The debug4jpf component runs JPF twice. The reason for this are breakpoints. As described in the previous chapters, JPF traces "all" possible traces of execution. In case there is a breakpoint at a particular point in the tested program, JPF would stop execution each time it would get to the breakpoint. This would not be acceptable for users as it can occur several hundreds or even thousands times per one test run. The proposed and implemented solution is running JPF twice. First run (further referenced as exploration run) finds a trace to the error and stores it into a temporary file. The second one (further referenced as debug run) uses the stored trace to steer the exploration directly to the error state. First run runs transparently to Eclipse and the user. The second one is controlled by the user-defined breakpoints and if error is found, the user is notified.

The exploration run uses its own listener, further referenced as error path listener. This listener handles only one event - propertyViolated. This event handler generates temporary file which contains program trace in JPF format (basically it contains instructions for the choice generators). This file is generated only if an error is found.

The debug run uses different listener, further referenced as debugging listener, and the file with the error path generated by the exploration run. To be more specific, in case the first run does not find any error, the file does not exist. This has the advantage, that the second run is not executed and the user is notified that there was no error found.

The debug4jpf component is started by Eclipse debugging plugin. As soon as the component is loaded, `started` event is fired (sent to Eclipse) and the processing is started. There are three ways how to finish debug4jpf - send exit command from Eclipse, process through both JPF runs to the end or in case there is no error found, debug4jpf finishes after the first run of JPF. In each case, the component fires `exited` event.

The exploration run of JPF is running at full JPF speed to get to the point of error and store the error path as quickly as possible. Full speed means that there is no additional logic in error path listener that would slow down the processing. The debug run of JPF is impacted by the performance of debugging listener. Its logic is responsible for breakpoints handling. This means that after each executed instruction it has to be checked if there is a breakpoint or not.

Both JPF runs are started and handled transparently to the user. It is presented to the user as standard Java program debugger interface.

The main difference between debugging standard Java program and debugging Java program through debug4jpf is stepping. In case of error, Eclipse (or any other IDE) users are used to step through Java programs line by line to check the program execution before the error occurs. In order to understand reasons why this is not supported by debug4jpf, it is necessary to realize that at the time of debug run of JPF, thread scheduler has been already preconfigured by exploration run of JPF. This means that stepping one line forward (or stepping into method invocation) in one thread could not be done without impacting other threads. It would cause confusion to the user as this is not standard way how stepping should behave. The user always controls only currently selected thread during standard Java program debugging. In JPF environment, this is not possible. An example is described in Figure 4.2. Executing one step in Thread 1 includes also three steps in Thread 2, terminating Thread 3 and starting new Thread 4.

Breakpoints are introduced to minimize impact on user comfort. They allow user to point to a line in a source code and whenever the debug run of JPF reaches this line, it suspends its execution (all threads are suspended). At this time all details of all live threads are available - threads, stack frames and variables. JPF processing continues after receiving `resume` command from the debugger.

Overall view on debug4jpf execution is depicted in Figure 4.3. To summarize, the listeners in debug4jpf handle the following events:

**Property violated** (JPF encountered a property violation) - it is handled by both listeners. The debugging listener handles it as execution suspended and Eclipse is notified. The error path listener handles it as an error path is created and stored in a temporary file.

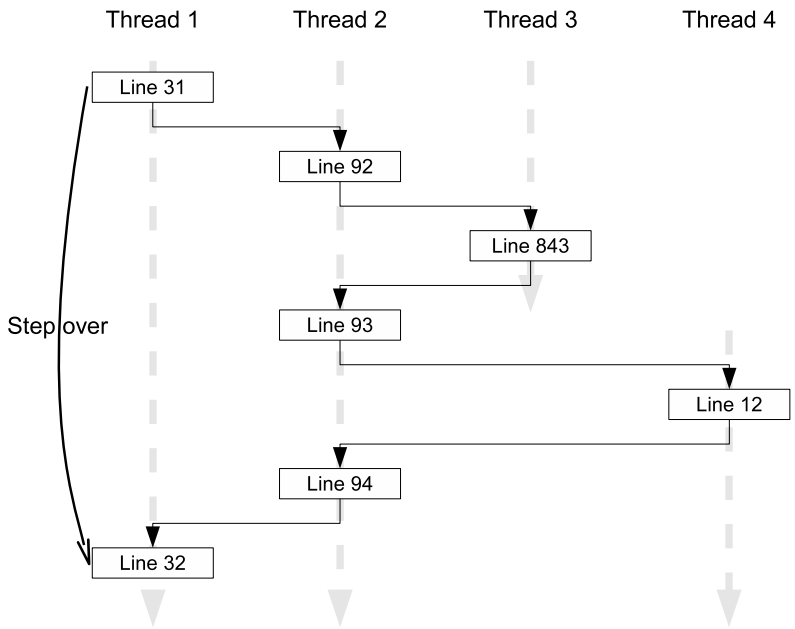



Figure 4.2: Thread scheduling example

**Execute instruction** (JVM is about to execute the next instruction) - it is handled by the debugging listener. Before each instruction is executed, it is checked whether there is a breakpoint enabled for this source file and line number. If there is a breakpoint, execution is suspended and Eclipse is notified.

### 4.2.1 Runtime Status Holder

JPF uses `Runtime` object to inform debug4jpf about its state and debug4jpf uses this object to request JPF state change. Possible states are:

**Not started** - debug4jpf has already started and is somewhere in the middle of initialization or execution of the exploration run of JPF searching for an error trace.

**Running** - debug4jpf has started, the exploration run of JPF is finished and the trace was written to a file. The debug run of JPF has already started and it is running.

**Set suspended** - Eclipse requested debug4jpf to suspend and debug4jpf is waiting for JPF to finish instruction execution to be suspended.

**Suspended** - there are three possible reasons for this state - Eclipse requested debug4jpf to suspend, or JPF encountered a breakpoint, or JPF encountered an error in the program under analysis.

**Set resumed** - Eclipse requested debug4jpf to resume. After JPF is resumed the state will be changed to running.

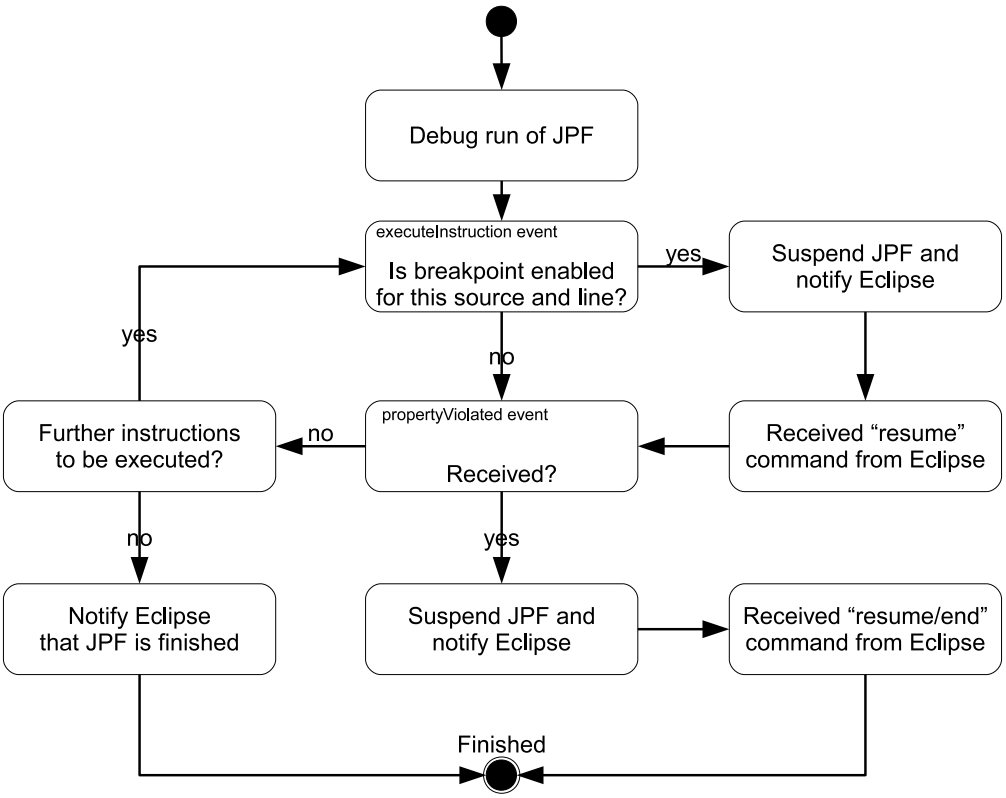



Figure 4.3: Debug run state diagram

**Set finished** - Eclipse requested debug4jpf to finish and debug4jpf is waiting for JPF to finish instruction execution to be finished.

**Finished** Both runs of JPF have already finished, debug4jpf is waiting to be exited.

State change diagram is depicted in Figure 4.4. Please note, that not all state change transitions are shown. The first reason is that the picture would be hard to read and the second is that these missed transitions should never be used while using this tool in the standard way. As an example, there should be a transition between states `Set suspended` and `Set finished`. However in the actual implementation, the state `Set suspended` is changed to `Suspended` right after current instruction is executed by JPF.

In order to satisfy Eclipse debugging interface and provide all details that user is used to while debugging Java applications, debug4jpf has to provide runtime details about threads, stack frames, variable names and their values. All these details are parts of JPF data model. Main root object is `JPF`. This object holds `Config`, `Search` and `VM` objects. The `Config` object contains configuration details. Initialization happens in a prioritized order, which means anything can be overridden from later configuration stages, all the way up to command line parameters.

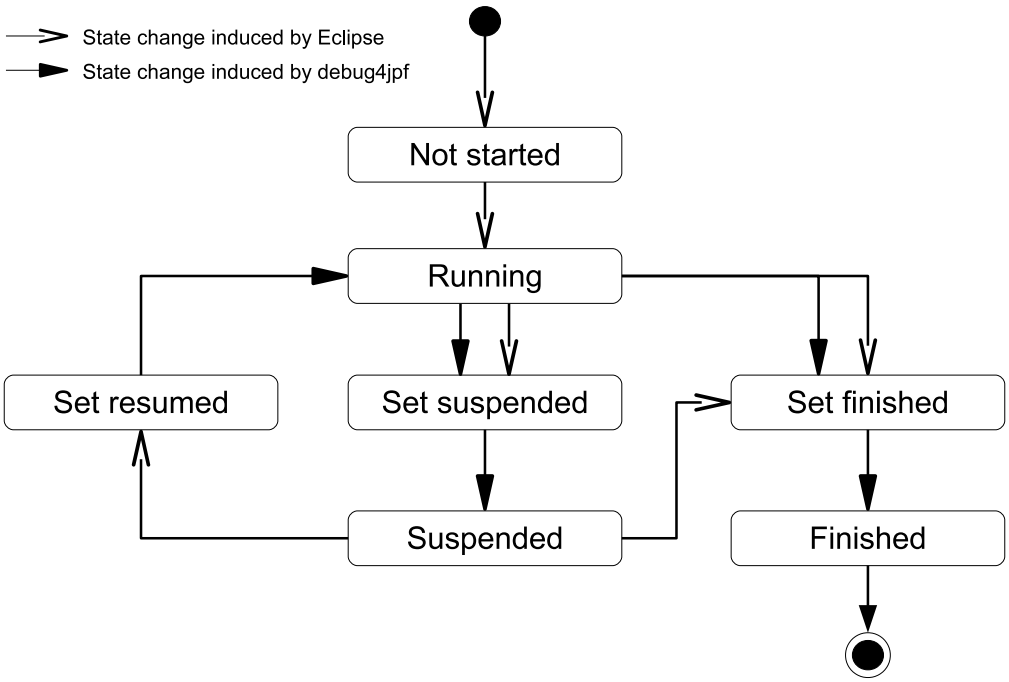



Figure 4.4: debug4jpf runtime state diagram

### 4.2.2 Data Model

The `Search` object provides details of an algorithm which JPF uses for tracing through the program that is being tested. An example of such detail is an execution path from the very beginning to the current state. This detail is used in exploration run of JPF to get the exact direct path to the error state.

The `VM` object holds all runtime information that is related to virtual machine execution. In this context, the virtual machine refers to virtual machine within JPF which executes instructions of a program that is being tested (not the virtual machine that executes JPF itself). The virtual machine object provides internal rather technical details of the execution. Objects which are referenced from `VM` can be divided into three categories - bytecode execution, object model and type + code management.

The bytecode execution group contains `SystemState` and `ThreadInfo` objects. The latter one is used in debug4jpf to get details of threads such as count of threads, thread state (e.g. running, blocked, terminated) and stack frames of the current thread. The `StackFrame` object is crucial for retrieving information about stack frames and variables names.

The stack frame and the variable name is used for getting a variable value. Objects from the object model group keep all object instances - fields, local variables and operands. These values are stored for each stack frame, therefore it is possible to reconstruct system state in each stack frame.

If there is a variable of non-primitive type, Eclipse have to know also the structure of this type - names and types of class fields. The type + code

management group of objects provides such details. `ClassInfo`, `MethodInfo`, and `FieldInfo` hold class, method and field structures. For classes, it is name, fields names and types, class methods etc.; for methods, it is name, parameters, return value etc.; for fields, it is name and type.

Figure 4.5 shows overall picture of JPF data model and relationships of particular entities. Gray entities are used in our approach.

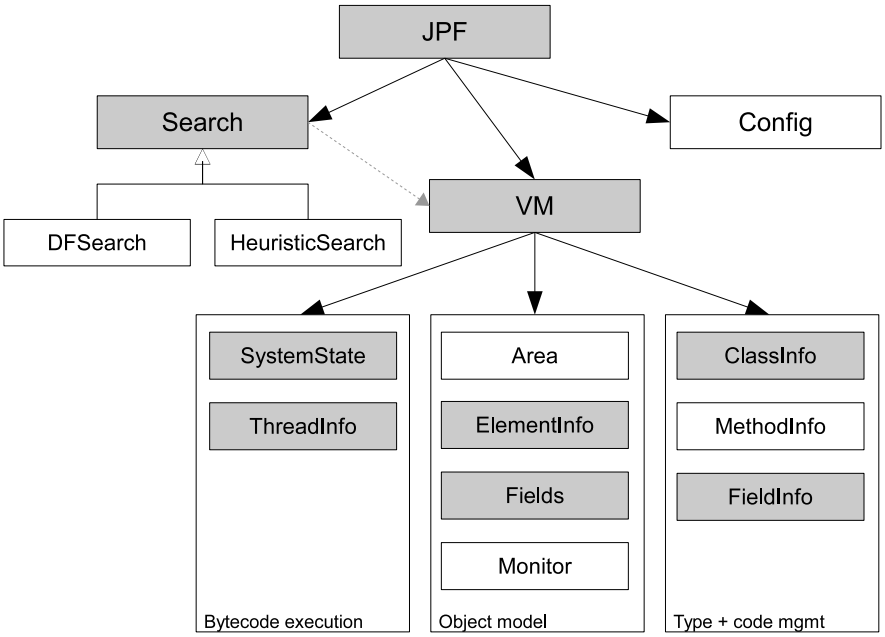



Figure 4.5: JPF data model

### 4.2.3 Commands and Events Handled by debug4jpf

The debug4jpf component is able to handle several commands. Generally, a source of these commands can be any external component which implements the communication protocol used by debug4jpf. In our approach, the external component is Eclipse with JPFDeb.core plugin installed. Commands are divided into three categories - runtime commands, variable commands and breakpoint commands.

Runtime commands are responsible for controlling debug4jpf processing. They start, suspend, resume and finish the processing of JPF and debug4jpf itself. Table 4.1 lists the supported runtime commands. The response to the runtime command is just notification that the command was accepted and it was queued to be processed.

Variable commands are responsible for threads, stack frames and variables retrieval. The sequence of commands starts with getting the list of all threads (running, blocked, terminated). Next step is getting stack frames (variable names) for all threads. This enables Eclipse to show thread list with links to the source code, i.e. their current positions defined by a file name and a line

| Command | Runtime status change | Result |
|---|---|---|
| start | `Running` | Status message |
| suspend | `Set suspended` | Status message |
| resume | `Set resumed` | Status message |
| exit | `Set finished` | Status message |

Table 4.1: List of the runtime commands supported by debug4jpf

number. The final step is to get values for all variables of the selected thread and stack frame. Table 4.2 lists all the supported variable commands.

| Command | Result |
|---|---|
| thread | List of the threads and their ids, names and states. |
| stack | List of the stack frames for the thread and their source file names, line numbers, method names and variable names. |
| var | Value for the variable in the stack frame of the thread. In case of object it returns also structure of the object. |

Table 4.2: List of the variable commands supported by debug4jpf

Breakpoint commands responsibility is to add and remove breakpoints. The debug4jpf component holds the list of all enabled breakpoints at runtime. Two commands in this category modify this list. Table 4.3 lists all the supported breakpoint commands.

| Command | Description | Result |
|---|---|---|
| set | Adds the breakpoint to the list of enabled breakpoints. | Status message |
| clear | Removes the breakpoint from the list of enabled breakpoints. | Status message |

Table 4.3: List of the breakpoint commands supported by debug4jpf

In addition to all commands, debug4jpf sends events to the source of commands - the debugger. These events notify the debugger that the requested change was successfully processed. It happens that the event is sent to the debugger even without previous command. This can occur in situations when processing hits a breakpoint (`suspended` event is sent), JPF gets to an error (`suspended` event is sent) or JPF gets to the end (`terminated` event is sent). Table 4.4 lists all supported events.

## 4.3 The JPFDeb.core Component

The user interface is implemented as a plugin to Eclipse. This enables easy integration and fully featured Java development and debugging environment. Plugins in Eclipse are used to extend the standard skeleton of Eclipse. Eclipse itself is the user interface application with the built-in plugin engine. All Eclipse

| Event | Description | Reason |
|---|---|---|
| started | The exploration run of JPF started | Response to the `start` command |
| suspended | The debug run of JPF suspended | Response to the `suspend` command or when breakpoint or error occurs |
| resumed | The debug run of JPF resumed | Response to the `resume` command |
| terminated | JPF run terminated | Response to the `exit` command or JPF gets to the end |

Table 4.4: List of the events supported by debug4jpf

features, e.g. Java editor, Java compiler, perspectives, views etc., are parts of plugins which come with the standard Eclipse distribution for Java developers.

The plugin, called JPFDeb.core, is used as the interface between the user and debug4jpf. This plugin is started during Eclipse initialization. There are several items added to the Eclipse menu and Run/Debug dialog window (Figure 4.6). Each time the user starts the program analysis by JPF, new instance of Eclipse process and debug target are loaded. The Eclipse process is a container for running the debug4jpf component. It starts new JVM instance and executes debug4jpf in it. The debug target is the main execution class of the plugin and the root element of Eclipse Debug framework data model (more details in Section 4.3.1). After JPF completes state exploration and debug4jpf is finished, the process and debug target objects are destructed.

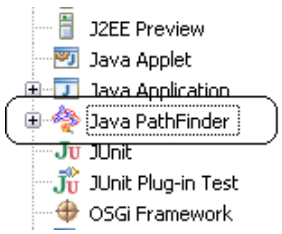



Figure 4.6: JPFDeb.core menu items and Run/Debug dialog

The JPFDeb.core plugin contains several extensions, which are the basic building blocks for Eclipse plugins. The `LaunchConfigurationTypes` extension configures available modes - run and debug. Run mode enables the user to analyze program with JPF and print out the JPF output. Debug mode enables user to fully use all features of JPFDeb.core. `LaunchConfigurationDelegate`, which is also configured in this extension, is instantiated by executing run/debug launch configuration. The delagate parses the launch configuration, starts the process and the debug target.

The `SourceLocator` and the `SourcePathComputer` extensions are responsible for locating a source code file name and line number. If a breakpoint is hit, these extensions find the file, open a text editor (the editor type is de-

pendent on the file type) and set the cursor to the line where the processing stopped.

The `Breakpoints` and the `Markers` extensions handle breakpoints. The `LineBreakpoint` is subclassed in order to support line breakpoints in our implementation. Breakpoints, which are inserted into/removed from source code, are communicated to the debug4jpf component.

### 4.3.1 Data Model

The JPFDeb.core component implements Eclipse Debug framework data model (overall picture in Figure 4.7). These classes are interacting with Eclipse via the debug target class. The debug target class instance is registered in Eclipse by `LaunchConfigurationDelagate`.

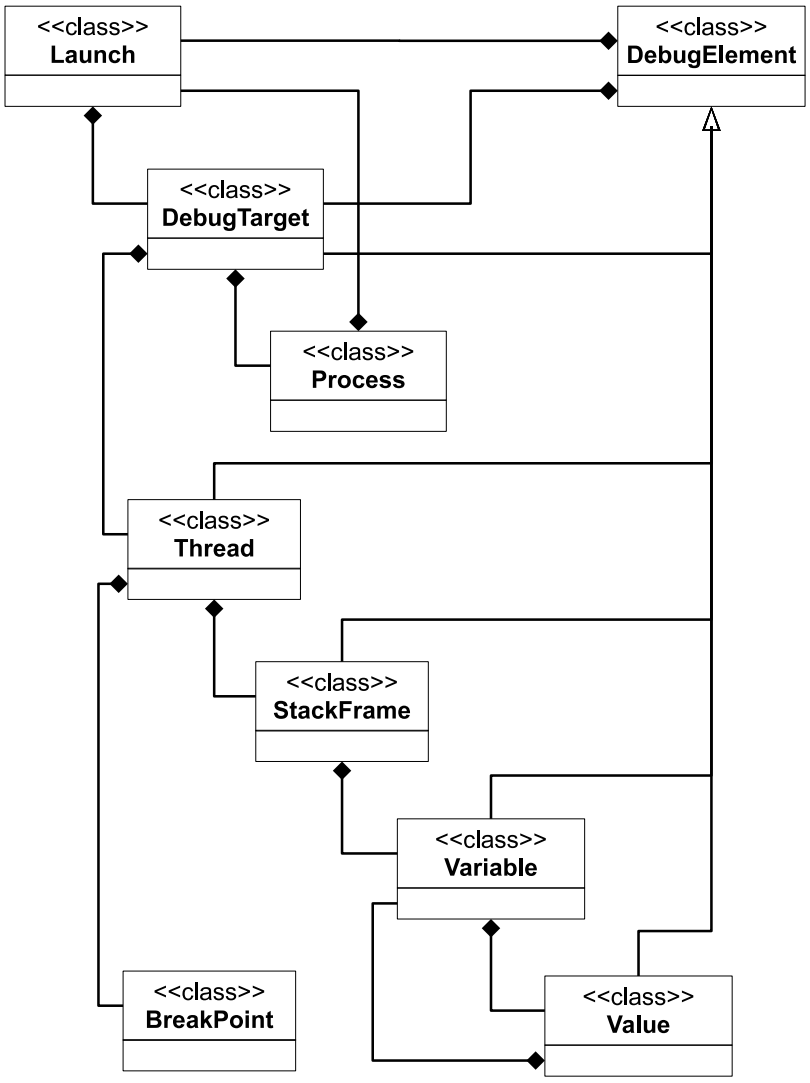



Figure 4.7: Eclipse Debug framework data model implementation

All other classes either directly or indirectly reference the Debug target UI element (marked as 1 in Figure 4.8). The Thread UI element (marked as 2 in Figure 4.8) instances are representing all threads currently existing in program exploration in JPF. Each thread has a list of Stack frames (marked as 3 in Figure 4.8). If a stack frame is selected, all variables are listed in Variable view. Every variable is represented by Variable UI element (marked as 4 in Figure 4.8) and its value is represented by the Value UI element.

The Breakpoint UI element (marked as 5 in Figure 4.8) instances represent the user defined breakpoints. Breakpoints are marked with a standard blue sphere and in case of breakpoint hit, the blue arrow is shown next to the line.

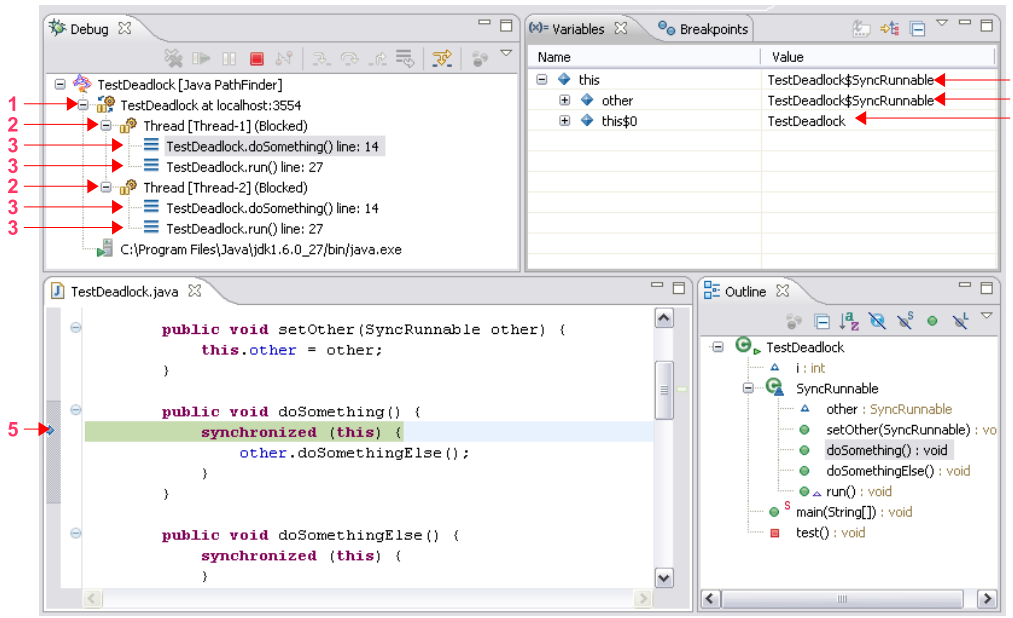



Figure 4.8: JPFDeb.core user interface components

## 4.4 Communication Protocol

Communication between debug4jpf and JPFDeb.core components is using an ad-hoc socket-based transfer protocol. The other options were serialization, RMI, and JPDA. The main reasons for not using any of the standards are:

- JPF does not provide client class instances - it has its own data structures which are used during verification
- Eclipse does not need object instances

In case of using one of the standards, two unnecessary transformations will be needed. One for fetching the object structure and data from JPF internal data structure and translating it to client class instance. The second one from client class instance to String representation which is used by Eclipse. This

was found unnecessary overhead and therefore it was decided to use String representation during the transportation. The socket-based transfer is natural consequence of using String as data representation. Characters with special meaning, e.g. newline, etc., are escaped in order to not spoil communication pattern.

There are two data flows defined - one for commands and another for the events flow. Each of these flows has one separate socket. The socket for commands is transfering data both ways - from the debugger to the debuggee (commands) and from the debugee to the debugger (responses). The socket for events is one-way only - from the debuggee to the debugger (events). Both sockets work with character strings as commands, responses and events.

Different situations require different communication types. All types of communication are described in Figure 4.9. Breakpoint handling commands are implemented in the request-response style (1). A command from the debugger is sent to the debuggee and a confirmation response is returned back. Variables handling commands use also the request-response style, but, unlike in the previous case, the response contains a payload (2), e.g. list of threads or stack frames. The third and most complicated communication style, which is used by the runtime handling commands, is using the request-response plus the notification pattern (3). As soon as a command is parsed, a response is sent synchronously back. After the command is handled properly, a notification (event) is sent from the debuggee to the debugger. The last communication style sends a simple event acknowledgement (4). This occurs in case debug4jpf changes its state based on an internal condition like a breakpoint hit or an error hit (not based on an external command). Regarding race conditions, the synchronized access to the sockets and the `Runtime` object makes sure that all the commands and events are processed in an expected order.

Error handling is based on various status codes in responses. If a command is processed successfully, the "OK" status code is returned. Otherwise, a particular error code describing what went wrong is returned.

List of all supported commands and events:

`start`

- this command does not have any parameter
- the receive confirmation is status message[1]
- the acknowledgement is `started` event

`suspend`

- this command does not have any parameter
- the receive confirmation is status message[1]
- the acknowledgement is `suspended` event

`resume`

- this command does not have any parameter
- the receive confirmation is status message[1]
- the acknowledgement is `resumed` event

[1]Status message format is `status <status>`, where <status> can be either "OK" or any string describing the status.

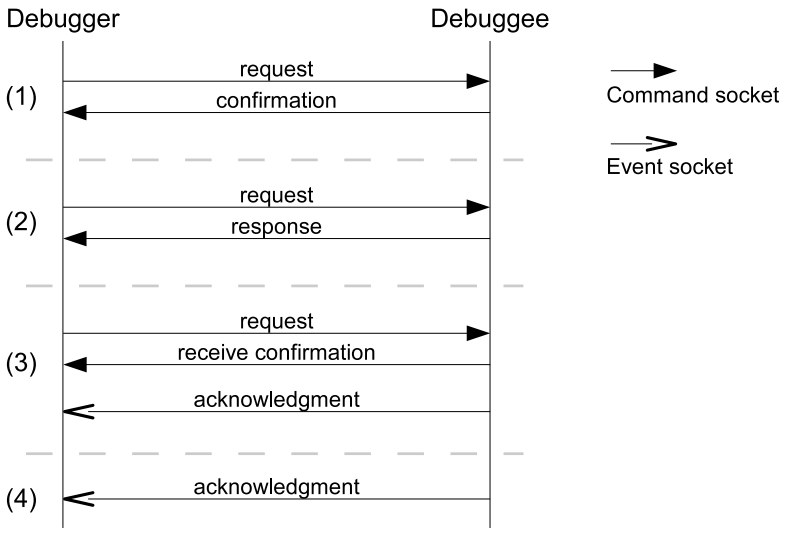



Figure 4.9: Communication styles

`exit`
- this command does not have any parameter
- the receive confirmation is status message[1]
- the acknowledgement is `terminated` event

`thread`
- this command does not have any parameter
- the response format is
  `<thread1Id>|<thread1Name>|<thread1Status>#<thread2Id>...`

`stack <threadId>`
- this command has one parameter - thread id
- the response format is
  `<sourceFile1>|<lineNumber1>|<methodName1>|<varName1_1>|`
  `<varName1_2>|...#<sourceFile2>...`

`var <threadId> <stackFrameId> <varName>`
- this command has three parameters - thread id, stack frame id, variable name
- the response format is
  `<varName>=<value>` for primitive types
  `<varName>=<varType>(<varName>.<varChildName1>,...)` for non-primitive types

`set <sourceFileName> <lineNumber>`
- this command has two parameters - source file name and line number where the breakpoint is located
- the confirmation is status message[1]

`clear <sourceFileName> <lineNumber>`

- this command has two parameters - source file name and line number where the breakpoint is located
- the confirmation is status message[1]

# Chapter 5

# Implementation

The previous chapter elaborates on the design of the approach. This chapter lists technical details. The first implementation decision was programming language and development environment. The JPF API as well as Eclipse plugins are implemented in Java, so Java was chosen as the programming language for debug4jpf and JPFDeb.core plugin. Programs that are tested by JPF are Java programs too. In order to simplify the front end part implementation in form of Eclipse plugin, Eclipse was chosen as the development environment for JPFDeb.core and debug4jpf. It supports many useful utilities and plugins that help with Eclipse plugins implementation.

Besides standard Java and Eclipse libraries, JPF is used as an external library. There were two options to link JPF to debug4jpf component

(i) include JPF in debug4jpf library

(ii) let user to link local JPF installation to the debug4jpf.

The first option was dismissed because it would not be able to reuse the JPF installation for any other purposes. Another reason is that debug4jpf is build as superior/control component on top of JPF and not as an JPF extension. The second option to let user link any local JPF instance turned out to be problematic. The development and issue fixing activities on JPF are not backward compatible. JPF data model API changed rapidly during debug4jpf implementation phase and the whole component stopped working after JPF had been updated. The final version of debug4jpf component was adapted to be fully compatible with the latest version of JPF[1]. This limits the versions of JPF, which are guaranteed to be fully supported by debug4jpf, to one. Following versions of JPF are not guaranteed to be fully compatible with debug4jpf.

## 5.1 Implementation Structure

The two components listed in Chapter 4 – debug4jpf and JPFDeb.core – are implemented as two separate Java projects. The JPFDeb.core project is

[1] At the time of working out the master thesis - revision 617+.

dependant on the debug4jpf one, as it instantiates debug4jpf. The dependencies are shown in Figure 5.1.

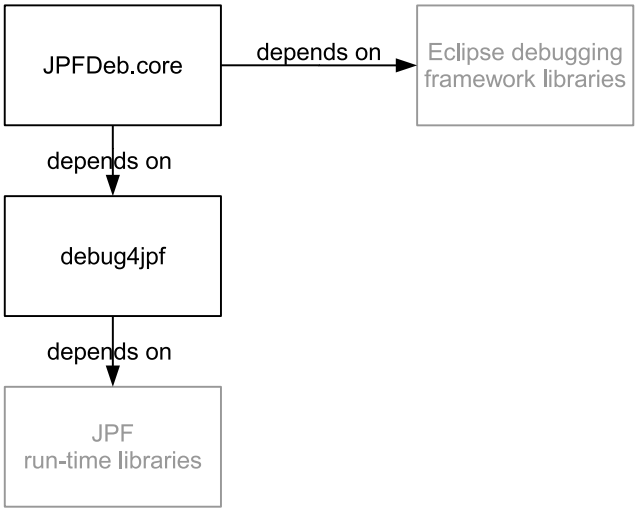



Figure 5.1: Projects and libraries dependencies

### 5.1.1 The debug4jpf Project Structure

The debug4jpf project is a standard Java project which is linked with JPF runtime libraries. This project is structured into the following packages:

**cz.cuni.mff.dsrg.debug4jpf.jpf** – This package contains classes responsible for instantiating JPF (exploration run and debug run).

**cz.cuni.mff.dsrg.debug4jpf.jpf.command** – This package contains classes that handle commands received from JPFDeb.core plugin.

**cz.cuni.mff.dsrg.debug4jpf.jpf.connector** – This package contains classes that handle network transfers.

**cz.cuni.mff.dsrg.debug4jpf.jpf.listener** – This package contains listeners for both runs of JPF.

**cz.cuni.mff.dsrg.debug4jpf.jpf.launcher** – This package contains the main class of the project which is used to start the debug4jpf component.

**cz.cuni.mff.dsrg.debug4jpf.jpf.util** – This package contains several utility classes which are used by classes from the previous packages.

After the Java source files are built into classes, they are packed to a single Java archive called `debug4jpf.jar`.

### 5.1.2 The JPFDeb.core Project Structure

The JPFDeb.core project is an Eclipse plugin project. Besides the standard Java builders there are additional builders for plugins - Plugin manifest builder, Extension point schema builder, and API analysis builder. All these builders prepare contents of the final plugin archive. This project is structured into the following packages:

**cz.cuni.mff.dsrg.jpfDeb.core** – This package contains classes for the plugin and the launch configuration delegate initialization.

**cz.cuni.mff.dsrg.jpfDeb.core.launching** – This package contains classes that support launch in run-time.

**cz.cuni.mff.dsrg.jpfDeb.core.model** – This package contains classes that implement the Eclipse Debug framework data model.

**cz.cuni.mff.dsrg.jpfDeb.ui.launching** – This package contains class that implements a dialog window with launch configuration from the user interface point of view.

**cz.cuni.mff.dsrg.jpfDeb.ui.model** – This package contains classes that support the data model on the user interface level.

The JPFDeb.core plugin is built into zip archive which contains two Java archives - one archive with the sources and one archive with the built classes.

## 5.2 Installation and User Guide

This section describes installation and usage instructions. Folders and files paths used in instructions are relative paths on the CD attached to this master thesis.

The only installation prerequisite is the proper Java 1.6 installation available on the target computer. Installation and configuration steps for users with the Windows operating system (for UNIX/Linux users, notes with differences are added to steps, if applicable):

1. Download and extract Eclipse v3.7.2 archive `eclipse_XXbit.zip`, which is located on the CD in the directory `/resources/Eclipse/` or the latest Java developer version of Eclipse can be downloaded from the internet[2], to a local directory (in further text referenced as `<ECLIPSE_DIR>`).
   Note: UNIX/Linux users should use `eclipse_linux_XXbit.tar.gz`.

2. Download and extract JPF r617+ release archive `jpf-core.zip` which is located on the CD in the directory `/resources/JPF/`, to a local directory (in further text referenced as `<JPF_DIR>`).
   Note: It is not recommended to download JPF from the internet as JPF Java API versions are highly non-backward compatible.

[2] www.eclipse.org

3. Copy the plugin archive `JPFDeb.core_1.0.0.jar` which is located on the CD in the directory `/resources/plugin/`, into the local directory `<ECLIPSE_DIR>/eclipse/plugins/`.

4. Run Eclipse by executing the `eclipse.exe` binary file located in the local directory `<ECLIPSE_DIR>/eclipse/`.
   Note: UNIX/Linux users should execute the `eclipse` binary file.

5. In the top menu, open menu item Window → Preferences → Run/Debug → String Substitution and modify "jpfHome" variable value to `<JPF_DIR>/jpf-core/`.

At this point, the JPFDeb.core plugin, the debug4jpf component and Java PathFinder are installed and configured for instant use.

### 5.2.1 Launch Configuration Setup

Eclipse supports two standard launch configuration types - run configuration and debug configuration. If the launch configuration delegate is able to handle both types, configuration of one type is automatically cloned to the other type. JPFDeb.core supports both types, therefore it is necessary to configure only one launch configuration. This configuration defines which launch configuration delegate should be used, which class in which project should be run and the input arguments (if applicable).

The basic JPFDeb.core configuration can be created by following these steps:

1. Create new configuration in the Java PathFinder launch group.

2. Write configuration name.

3. Select Java project in which the main class of the tested program is located.

4. Select the main class of the tested program.

5. (Optional) Specify the file name of application specific property file (relative path to the project root folder and the file name must have ".jpf" extension)

The basic configuration example is depicted in Figure 5.2. The main class `TestDeadlock` (marked as 1) from project `TestDeadlock` (marked as 2) can be checked by JPF by running the launch configuration called "Test deadlock" (marked as 3). In the Arguments tab (marked as 4), there is a possibility to specify an application level property file. This can be done by providing the relative path (based in the project directory) to the property file in the Program arguments field.

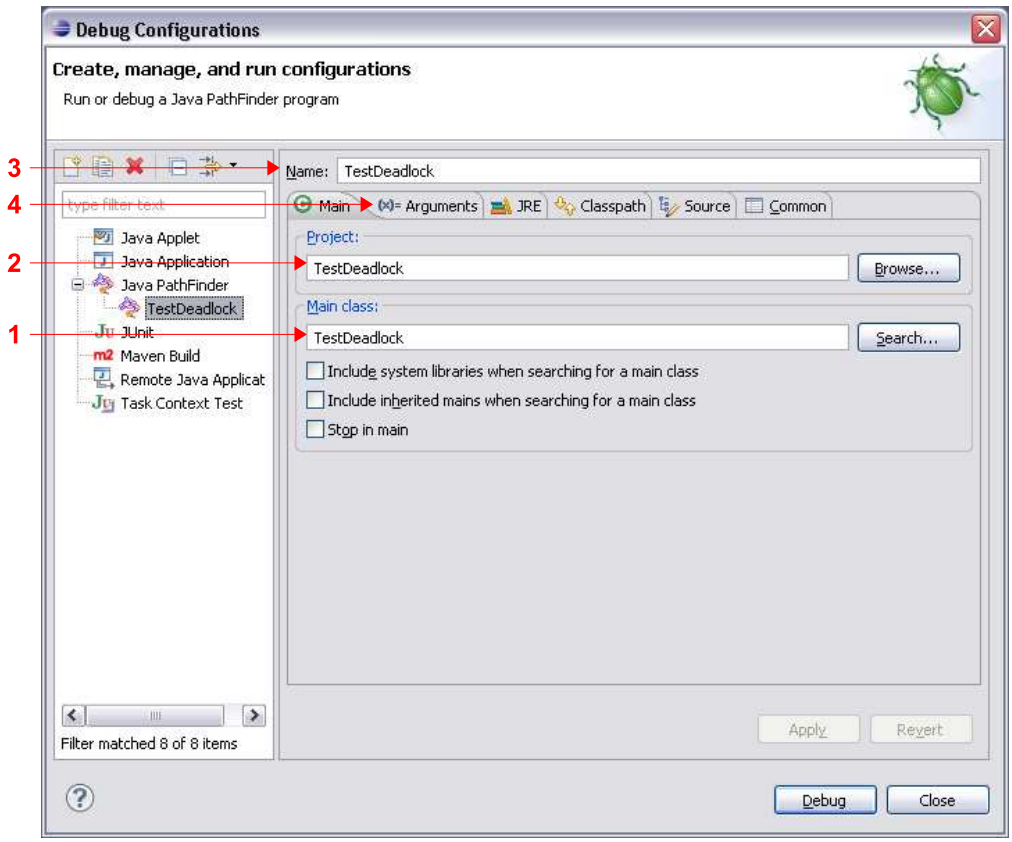



Figure 5.2: JPFDeb.core launch configuration

### 5.2.2 Model Checking in Run Mode

In case the user just want to see the standard output of JPF after the program is checked, then run mode should be used. The JPFDeb.core plugin in run mode starts JPF and prints out the standard JPF output. No enhanced features are active.

Custom JPF properties, e.g. listeners, checked constraints, etc., are supported. The project specific `<name>.jpf` file has to be available within the project directory and configured as a program parameter in Run/Debug dialog. To avoid overwriting changes in JPF configuration properties done by the debug4jpf, it is recommended to put only those JPF properties into property file, that the user want to change.

### 5.2.3 Model Checking in Debug Mode

On the other hand, if JPFDeb.core is started in debug mode, all enhancements are active and the user can exploit all advantages of this plugin.

Figure 5.3 shows Eclipse workbench right after the program checker is started. There are two important views - Debug (marked as 1) and Console (marked as 2). The Debug view is listing all currently running threads. The

Console view shows the standard JPF output which is generated by the current JPF run.

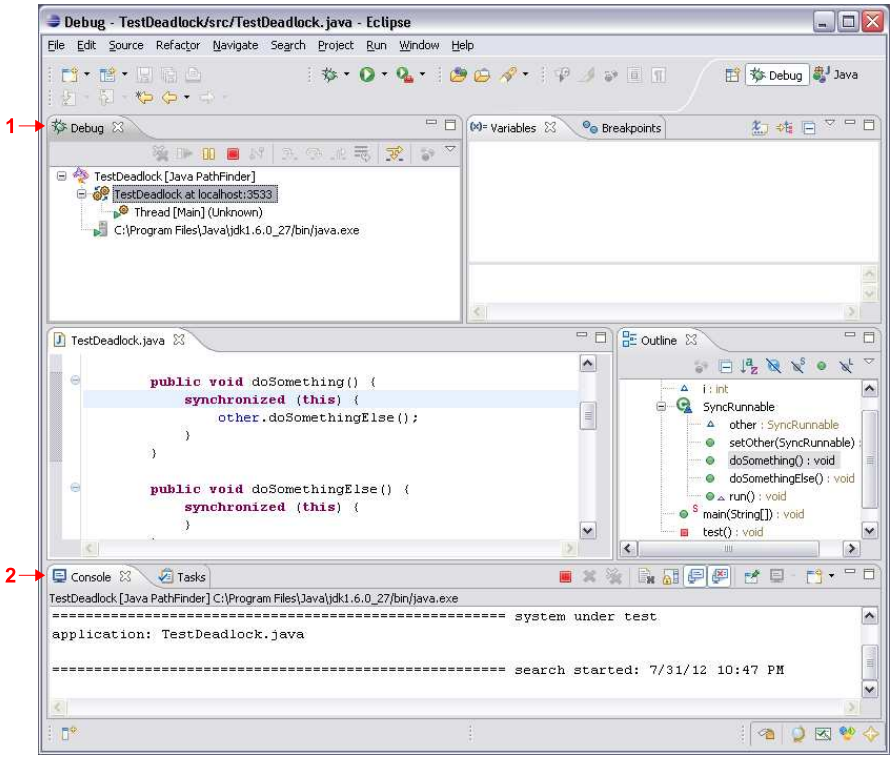



Figure 5.3: Starting the model checker

As soon as JPF finds an error (if there is any), the processing stops. Figure 5.4 shows the Eclipse workbench right after an error is detected. The Debug view lists all threads (marker as 1) with their states and after extending the tree node representing thread, stack frames (marked as 2) are listed. The user can list all variables linked with the stack frame by selecting one in the tree. The variables (marked as 3) are listed in the Variable view. Each stack frame links also the actual source file name and line number. This file is opened in Java editor (marked as 4) and cursor is placed on the actual line (marked as 5).

When the user finishes the current state exploration, plugin processing can be resumed with the Resume button. Finally, the plugin lets the JPF, the debug4jpf and the Eclipse launch finish. All JPF output is written to the Console view, so the user is not deprived of this standard JPF feature that he/she is used to.

There is one difference in controlling the debug launch execution. While the user is allowed to control each thread separately, it is not possible in the JPF environment. The reasons for this are listed in Section 4.2. This means that resume operation can be executed only on the level of debug target and not on the thread level.

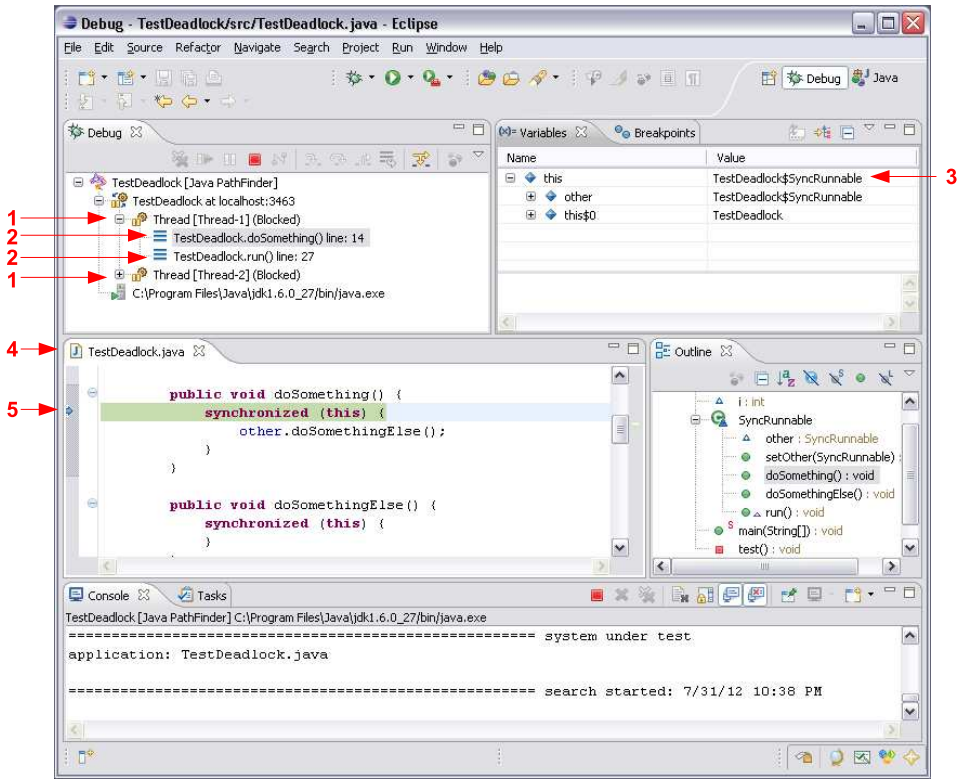



Figure 5.4: The model checker found an error

### Breakpoints

The breakpoints are the only control mechanism, that can be used to suspend th JPF states exploration. The step over and step in debugging features are not available for the reasons listed in Section 4.2.

As soon as the debug run of JPF hits the source file and line number on which a breakpoint is enabled, the state exploration is suspended and the control is passed to Eclipse. At this point, the workbench is behaving the same way as it behaves in case of an error hit.

## 5.2.4 Sample Projects

Several sample projects are located on the attached CD. They can be used for quick tests of how our approach behaves in various situations.

They can be found in the directory `/samples/` on the attached CD. The list of projects:

- TestDeadlock
- DiningPhilosophers
- NumericValueCheck
- Racer

- Random
- StopWatch

These sample projects are originally part of the JPF project and should present the capabilities of JPF. They are reused also for the presentation purposes of our approach capabilities.

In order to run these projects, the user has to import the Eclipse project into the workspace and then continue with the standard procedure described in the prior parts of this section.

# Chapter 6

# Related Work

This chapter lists few of the projects which are related to this thesis. Their features list is a subset of the features list of our approach, they are extending the JPF output capabilities, but in a different way with different goals, or they are applying the similar approach but for a different tool in a different environment.

## 6.1 Eclipse-jpf

The eclipse-jpf [19] is an Eclipse plugin which is able to run JPF on selected application level property file. This plugin is a minimal implementation which supports just starting an external JPF process from within Eclipse.

The main and only features are:

- start the JPF shell [20] in an external process and provide selected configuration to it
- receive requests from JPF shell to place the cursor on the line in the editor for corresponding source file

This project was created in parallel to this thesis. Nowadays it became a favourite tool for users which use Eclipse (for Netbeans users there is also an alternative for Netbeans available). It makes it much easier to run the JPF instance, however, the project does not go further – start JPF with GUI, wait for the execution to finish, and terminate.

The eclipse-jpf project and our approach are doing the same thing to a certain extent. Based on this fact, it would be good decision to integrate our approach into eclipse-jpf project, which was already published to the users and is used by wide JPF community. This would make our solution an extension of eclipse-jpf and not delivering another solution with similar features.

## 6.2 JPF Inspector

JPF Inspector [21] is a tool which is used for

**JPF execution inspection** – explore and modify the program state (threads, call stacks, and heap objects)

**JPF execution control** – breakpoints and stepping (forward and backward) at different levels of granularity

In addition to standard Java debuggers, JPF Inspector supports an explicit control over thread scheduling.

The main features of the JPF Inspector tool are:

**Breakpoints** – multiple types of breakpoints are supported – a specific line of code, instruction at a given position, specific instruction type, read or write access of a variable, object creation and destruction, garbage collection, transition boundary. New breakpoint state "log" is introduced. If the program hits a breakpoint in the log state, a log message is printed out without interrupting the execution.

**Single-step execution** – it is possible to step through the JPF execution. The step can be executed in both ways – forward and backward. JPF Inspector can be configured to allow the user to select the next choice at each state.

**Program state inspection** – the user can explore the program state, heap objects and call stacks of threads. JPF Inspector uses a custom expression language for this purpose.

**Program state modification** – the program state can be modified with limitations. Then the user can simply check how the modifications influence the behavior of the program.

**Record and replay** – all commands performed in the current session can be saved into a file and replayed later.

**Dynamic assertions** – conditional breakpoints located in the source code can be loaded dynamically.

JPF Inspector provides a custom user front end. This GUI is implemented as a new panel for the JPF shell framework [20].

The main difference between JPF Inspector and our approach is that while JPF Inspector is mainly focused on the execution period, the combination of JPFDeb.core and debug4jpf is focused on the result interpretation and insight. Other difference is the user interface – JPF specific jpf-shell vs. general well-known Eclipse.

## 6.3 CHESS

The CHESS project [11] developed by the Microsoft Research Group is strictly limited to searching for errors related to the parallelism in concurrent Win32 programs. It is capable to replay the thread schedule which led to the error

state. In this case, the testing and debugging environment is Microsoft Visual Studio.

If compared to JPF, it supports only limited number of rescheduling. If the error would occur only in case of larger number of reschedules, it would not be found by CHESS. JPF does not have the limitation on this number.

CHESS is not simulating, but executing the program during the verification. This implies that CHESS has no visibility on the program state. If there is an infinite loop in the program, CHESS will never finish its verification. JPF would found out that this program state was already visited and it would consider this execution trace as explored.

This approach would be good alternative to our approach in .NET environment, if it would not be so limited (Win32 programs, validation only for concurrent errors, and limited number of rescheduling).

## 6.4 Counterexample as Executable Program

The approach in [22] is quite different from all of the above. It lets the verification system (Simplify [23] or Z3 [24] in this case) to find the counterexample for which the validation fails. Then the executable program for this counterexample is generated and can be run and debugger by the user.

Three attributes of the executable:

- it can be debugged within the standard debugger to analyze the counterexample
- specification and program runtime checks errors are included
- runtime checks for all specifications are performed in order to allow the user to detect spurious errors

The constructed counterexample program is simulating the behaviour of the tested program, but it is not the tested program itself. This raises a concern that if the user is not debugging the tested program itself, but some generated program, then it might not be easy enough to find the connection between the original and generated program.

In JPF and our approach, the tested program itself is being simulated as well as inspected via debugging interface.

On the other hand, .NET framework is well-known programming language and it can be debugged by various debuggers and the selection is only on the user.

# Conclusion

The main goals of this thesis were to propose and implement new debugging interface to JPF, which would enable the visualization of all necessary details to easily interpret the JPF results.

The proposed approach was to use Eclipse, the widely spread development environment, as a container for the debugger part of the implementation. There were two subprojects designed and implemented – JPFDeb.core and debug4jpf.

The former one is an Eclipse plugin which display all the details to the user. It uses Eclipse Debug framework for integration into the Eclipse development environment.

The latter one is responsible for controlling and communicating with the JPF instance. The listener is used to control JPF state exploration. This listener is notified each time an event occurs. This lets the debug4jf component inform the Eclipse instance about the state exploration status.

By utilizing this new debugging interface to JPF, the user gets the possibility to receive the JPF results in a visual form. In case of hitting an error, there is a tree view with threads and stack frames as nodes. The user can select any tree item to see the details. If the stack frame node is selected, the list of variables and their values is shown.

All these new features provides the user with the understandable, clear and well-known (same as Java debugging) interface to JPF.

# Bibliography


[1] E. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetry in Temporal Logic Model Checking. In *Proceeding of the 5th Internation Conference for Computer-Aided Verification*, volume 697 of *LNCS*, 1993.

[2] E. Emerson and A. Sisla. Symmetry and Model Checking. In *Proceeding of the 5th International Conference for Computer Aided Verification*, volume 697 of *LNCS*, 1993.

[3] C. Ip and D. Dill. Better Verification Trough Symmetry. In *Proceedings of the 11th International Symposium on Computer Hardware Description Languages and their Application*, North Holland, 1993.

[4] E. Clarke, E. Emerson, S. Jha, and A. Sistla. Symmetry Reductions in Model Checking. In *Proceedings of the 10th International Conference for Computer-Aided Verification*, volume 1427 of *LNCS*, 1998.

[5] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Preceedings of 6th International Conference on Computer Aided Verification*, volume 1254 of *LNCS*, 1997.

[6] H. Saidi. Modular and Incremental Analysis of Concurrent Software Systems. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, pages 92–101, 1999.

[7] S. Das, D. Dill, and S. Park. Experience with Predicate Abstraction. In *Preceedings of the 11th International Conference on Computer Aided Verification*, volume 1633 of *LNCS*, 1999.

[8] H. Saidi and N. Shankar. Abstract and Model Check while you Prove. In *Proceedings of the 11th International Conference on Computer Aided Verification*, volume 1633 of *LNCS*, pages 443–454, 1999.

[9] M. Colón and T. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Proceedings of the 10th Internation Conference on Computer Aided Verification*, volume 1427 of *LNCS*, 1998.

[10] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10(2), April 2003.

[11] M. Musuvathi, S. Qadeer, and T. Ball. CHESS: A Systematic Testing Tool for Concurrent Software. Technical report, Microsoft Research, 2007.

[12] P. Jančík, P. Parízek, and J. Kofroň. Advanced Debugging with JPF Inspector, 2011.

[13] Peter Mehlitz. jpf-aprop. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-aprop.

[14] C. Artho. jpf-trace-server. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-trace-server.

[15] P. C. Mehlitz, W. Visser, and J. Penix. *The JPF Runtime Verification System*, 2005.

[16] D. Gruntz. Java Design: On the Observer Pattern. University of Applied Sciences, Aargau.

[17] D. Wright and M. Rennie. The Eclipse Debug Framework. In *Proceedings of the EclipseCon 2006 conference*. IBM, 2006.

[18] O. White. JAVA EE Productivity Report 2011. Technical report, ZEROTURNAROUND, January 2011.

[19] S. Badame and P. Mehlitz. eclipse-jpf. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/eclipse-jpf.

[20] S. Badame and P. Mehlitz. jpf-shell. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-shell.

[21] P. Jančík and P. Parízek. JPF Inspector. http://babelfish.arc.nasa.gov/trac/ jpf/wiki/projects/jpf-inspector.

[22] P Müller and J. N. Ruskiewicz. Using Debuggers to Understand Failed Verification Attempts, 2011.

[23] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for programc checking. Technical report, HP Laboratories, Palo Alto, 2003.

[24] L. de Moura and N. Bjorner. Z3: An efficient SMT solver. In *TAGAC*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

# List of Figures

# List of Tables

# Appendix A

# CD Contents

The contents of attached CD have the following structure:

```
/
├── javadoc/
│   ├── debug4jpf/
│   └── JPFDeb.core/
├── resources/
│   ├── Eclipse/
│   ├── JPF/
│   └── plugin/
├── samples/
│   ├── DiningPhilosophers/
│   ├── NumericValueCheck/
│   ├── Racer/
│   ├── StopWatch/
│   └── TestDeadlock/
├── sources/
│   ├── debug4jpf/
│   └── JPFDeb.core/
├── thesis/
└── README.txt
```

The directory descriptions are listed in the following table.

| Path | Content description |
|---|---|
| /javadoc/debug4jpf/ | Contains generated JavaDoc documentation for the debug4jpf project. |
| /javadoc/JPFDeb.core/ | Contains generated JavaDoc documentation for the JPFDeb.core project. |
| /resources/Eclipse/ | Contains Eclipse v3.7.2 release packages for 32bit/64bit Windows/Linux OS. |
| /resources/JPF/ | Contains JPF r617+ release archive. |
| /resources/plugin/ | Contains JPFDeb.core plugin which includes also debug4jpf archive. |
| /samples/DiningPhilosophers/ | Contains the sample Eclipse project which demonstrates deadlock error. |
| /samples/NumericValueCheck/ | Contains the sample Eclipse project which demonstrates range check error. |
| /samples/Racer/ | Contains the sample Eclipse project which demonstrates race condition error. |
| /samples/Random/ | Contains the sample Eclipse project which demonstrates random values usage within the program. |
| /samples/StopWatch/ | Contains the sample Eclipse project which demonstrates exploration of off-nominal paths. |
| /samples/TestDeadlock/ | Contains the sample Eclipse project which demonstrates deadlock error. |
| /sources/debug4jpf/ | Contains the source code of the debug4jpf project. |
| /sources/JPFDeb.core/ | Contains the source code of the JPFDeb.core project. |
| /thesis/ | Contains the PDF version of the master thesis. |
| README.txt | Contains the directories description similar to this one. |